

# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**



Mark Andreessen  
Renowned VC

Software is eating the world,  
in all sectors.

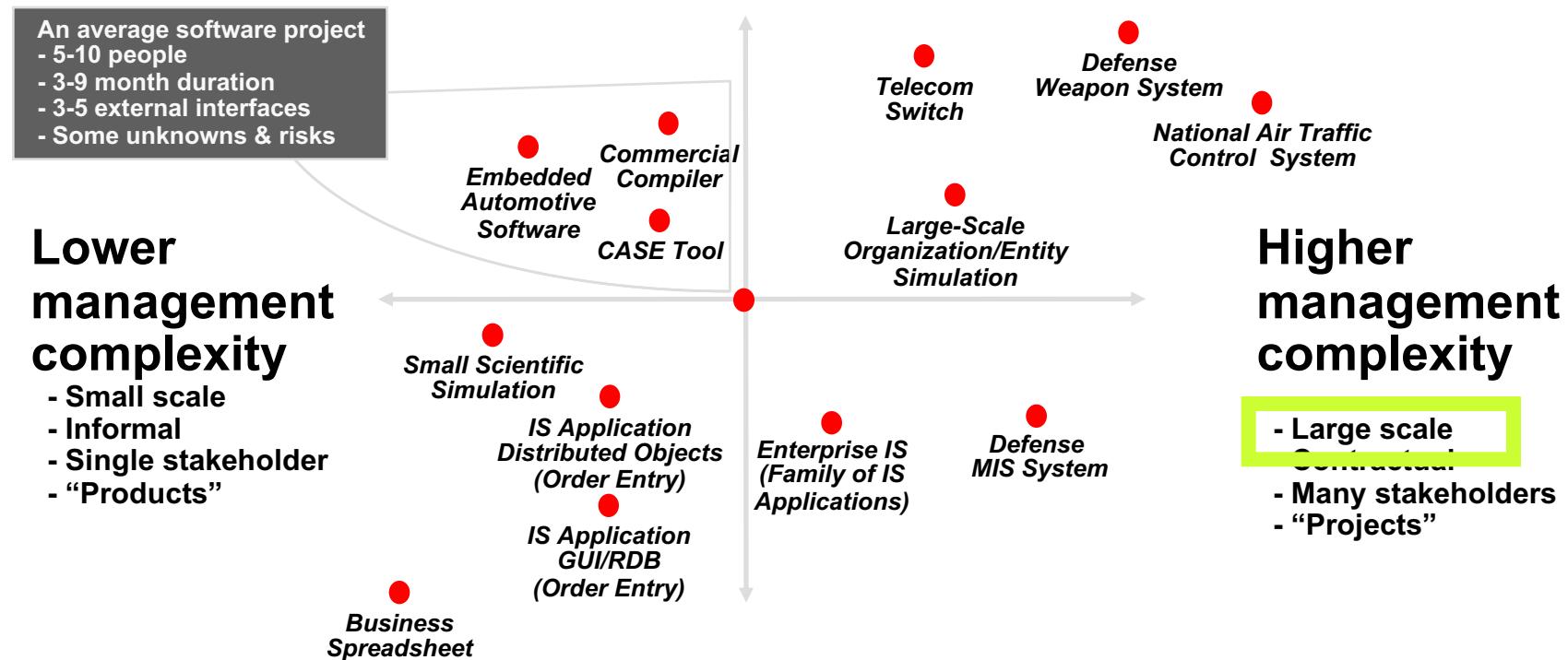
"The Wall Street Journal" in 2011

In the future  
every company will become  
**a software company**

# Software Complexity

## Higher technical complexity

- Evolutionary, iterative, distributed, fault-tolerant
- Custom, unprecedented, architecture reengineering
- High performance



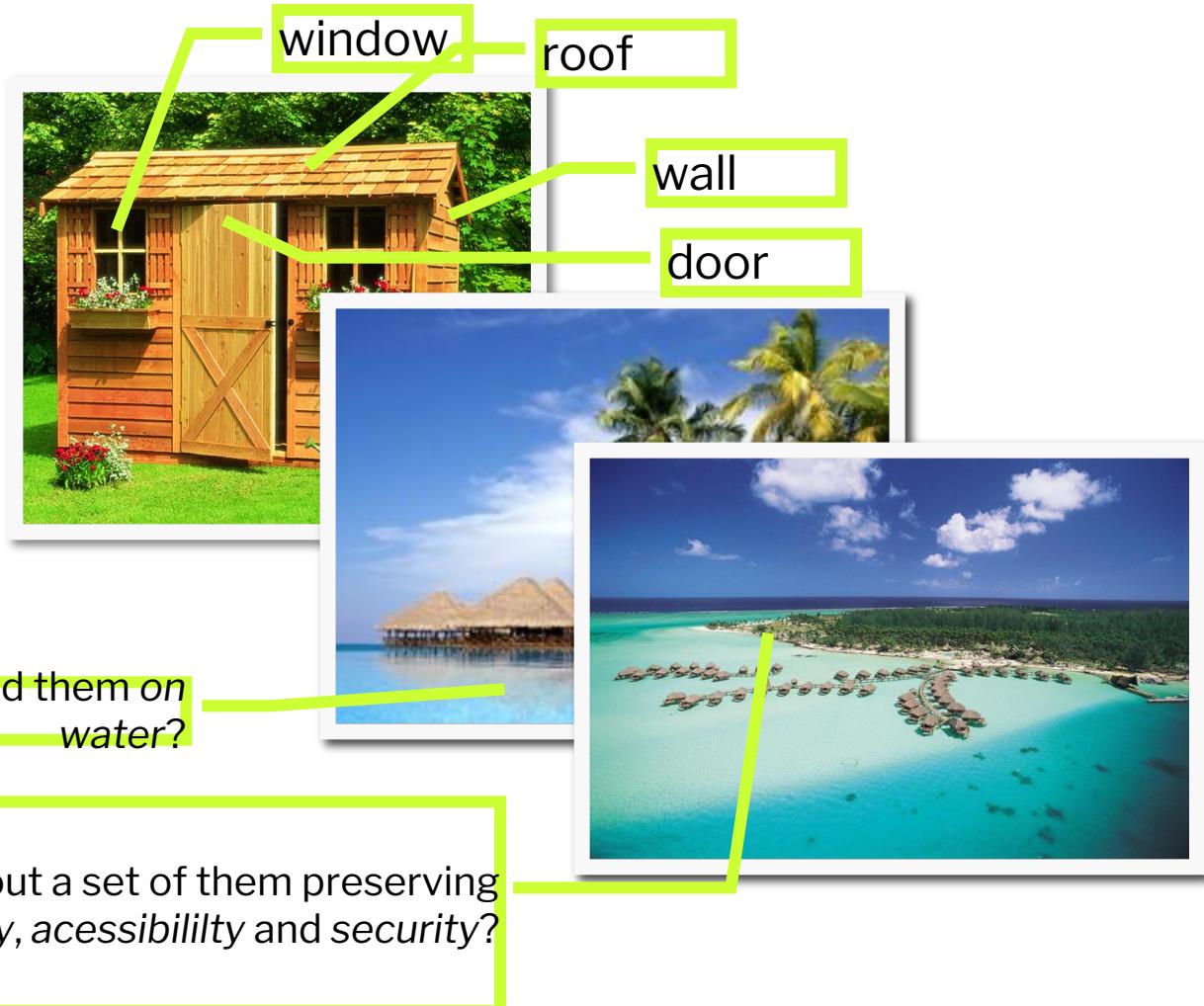
## Lower technical complexity

- Mostly 4GL, or component-based
- Application reengineering
- Interactive performance

Walker Royce

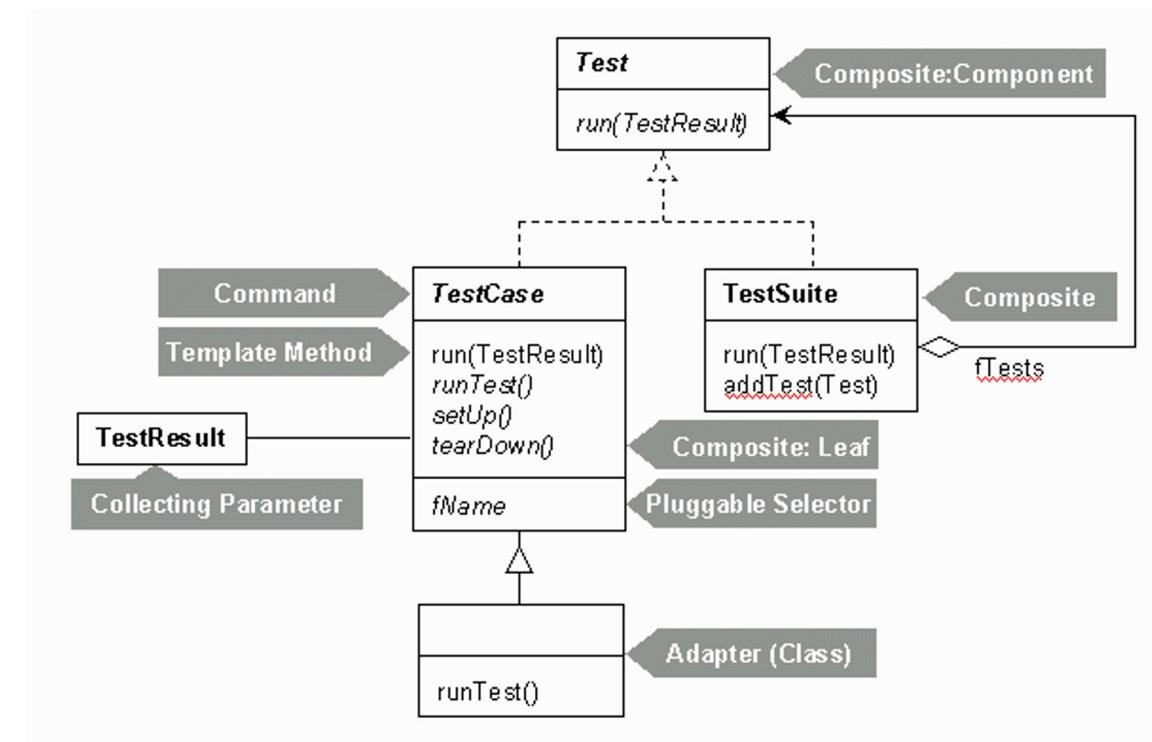
# Architecting...

To architect a small house “seems” to be very easy...



# Architecting JUnit...

“Voilá” a well known “little software house”: xUnit



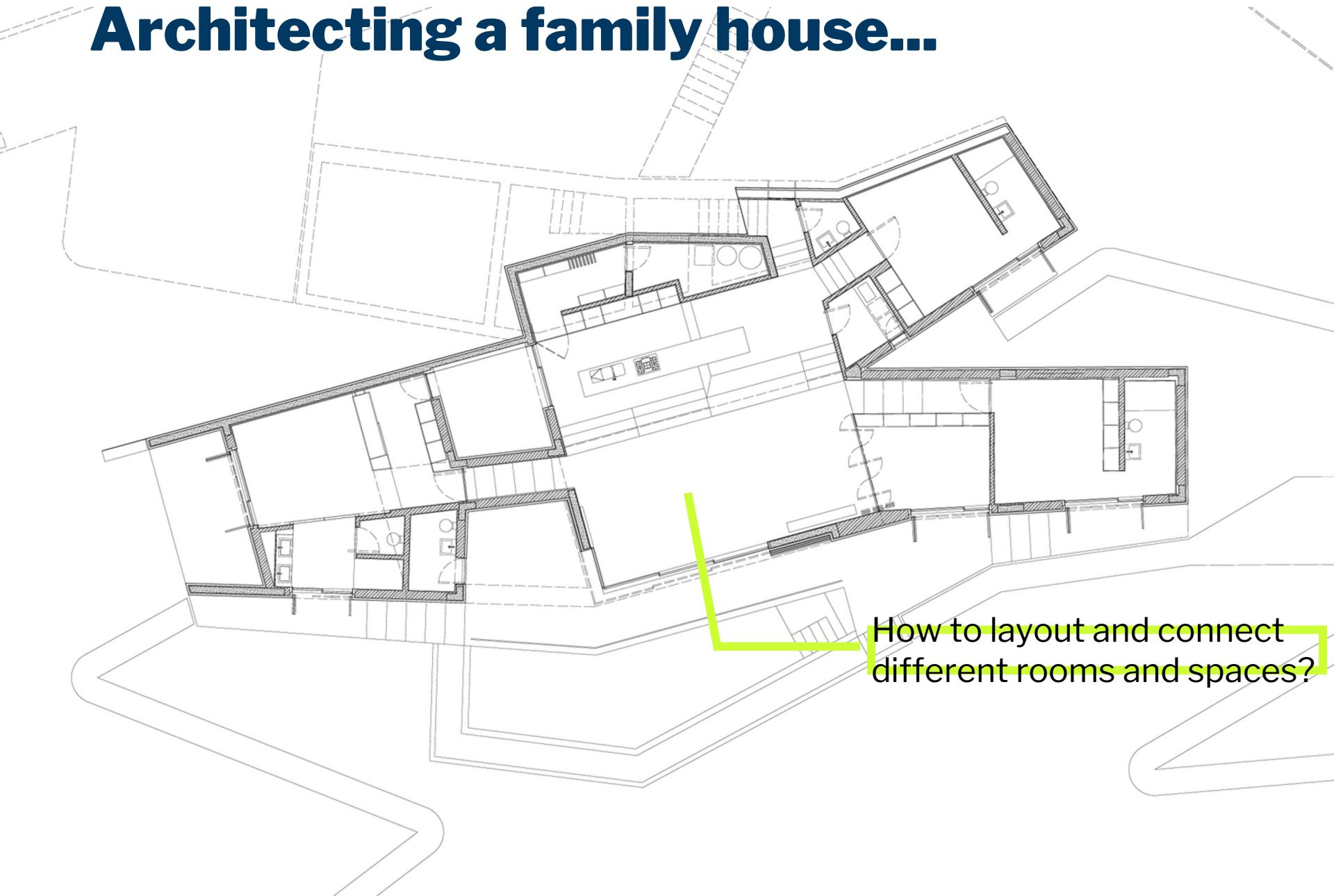
Which building elements can we identify here? Hmm...

# Architecting a family house...



How to layout and connect different rooms and spaces?

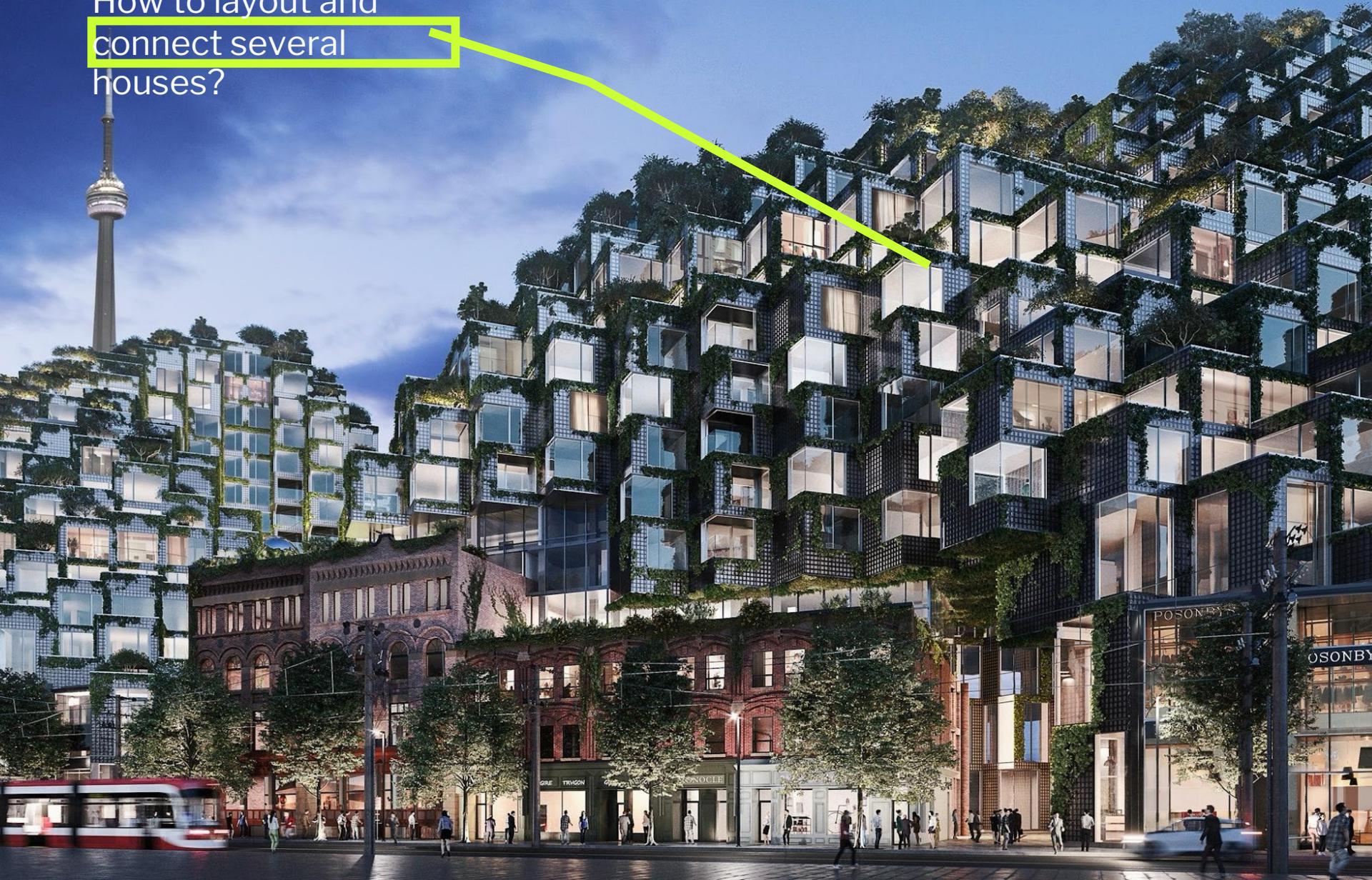
# Architecting a family house...



How to layout and connect  
different rooms and spaces?

# Architecting a multi-story building...

How to layout and  
connect several  
houses?



# Architecting a multi-story building...

How to layout and connect several houses?





How to build  
and maintain  
such building,  
and make it  
live?

# Software cannot fail!!!



A photograph of a long, brightly lit corridor in a data center. Both sides of the corridor are lined with tall, white server racks. The floor is made of large, light-colored tiles. The ceiling is white with a grid of fluorescent light fixtures. In the distance, there are red exit signs above doors. On the right side, there is a blue wall and a row of smaller, light-colored doors or panels.

**Software cannot fail!!!**

# Architecting...

Main differences between all these buildings:

Scale

Cost, Time, Teams

Process, skills

Materials and technologies

Risks

Robustness

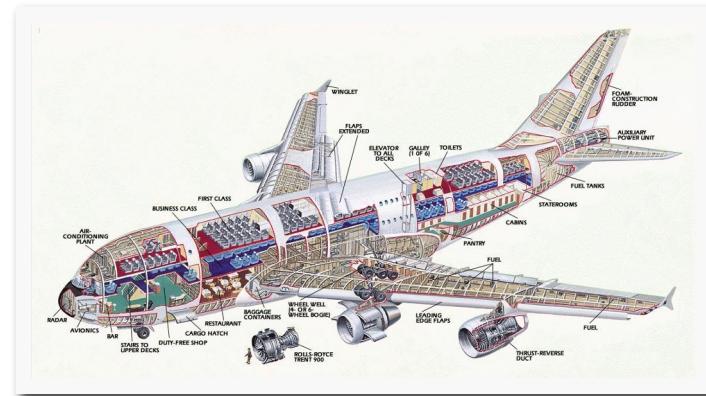
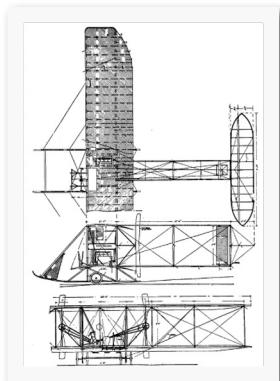
Longevity...

# Traditional Engineering

Traditional engineering have evolved based on “stable systems”, mature, good references

Stable systems have evolved over time

- By trial and error
- By constant adaptation and specific needs
- By reuse and refinement of well-proven solutions
- By several advances in materials, processes and standards



# Software Construction: is it harder? Yes!

Invisible constructions (logical product)

Temporal nature of software is complex and hard to understand (discrete systems)

Big needs to change and evolve along lifespan

Software must be very easy to adapt; each case is a case

Underlying technologies evolve very fast

Short history (since 1960?)

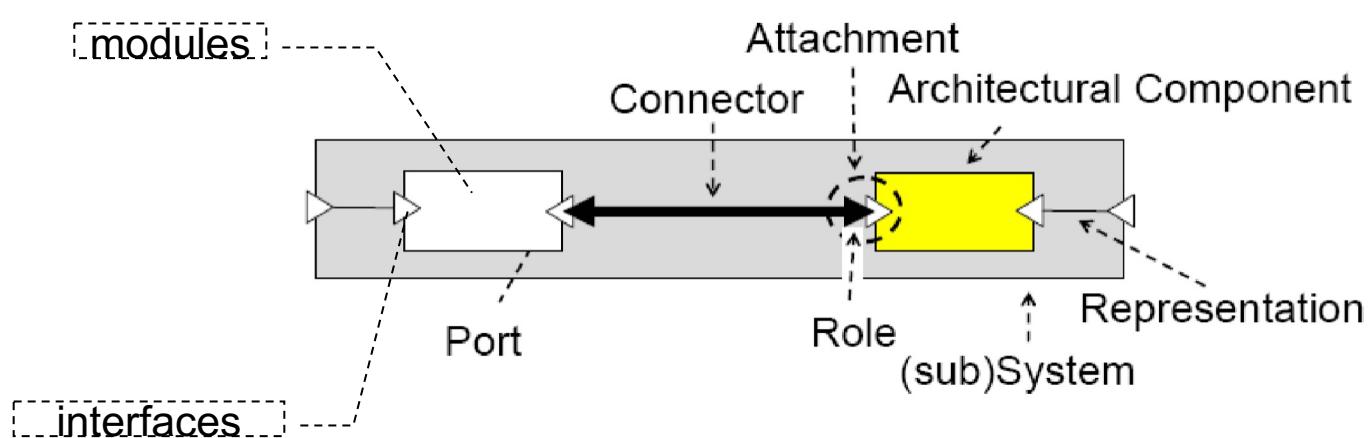
No “nature laws” or “physical laws” to conform to...

# Software Architecture

Software Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

*IEEE Recommended Practice for Architectural Description of Software Intensive Systems (IEEE 1471-2000)*

- Which **structural elements** and **interfaces** is the system made of?
- Which **composition mechanisms** are used to compound elements?
- Which **collaboration behaviors** between elements?
- Which main **architectural style** informs the system?



# The Software Architecture Role

## Architectural Drivers

Understanding the goals; capturing, refining, and challenging the requirements and constraints

## Designing Software

Creating the technical strategy, vision and roadmap

## Technical Risks

Identifying, mitigating and owning the technical risks to ensure that the architecture “works”

## Architecture Evolution

Continuous technical leadership and ownership of the architecture throughout the software delivery

## Coding

Involvement in the hands-on elements of the software delivery

## Quality Assurance

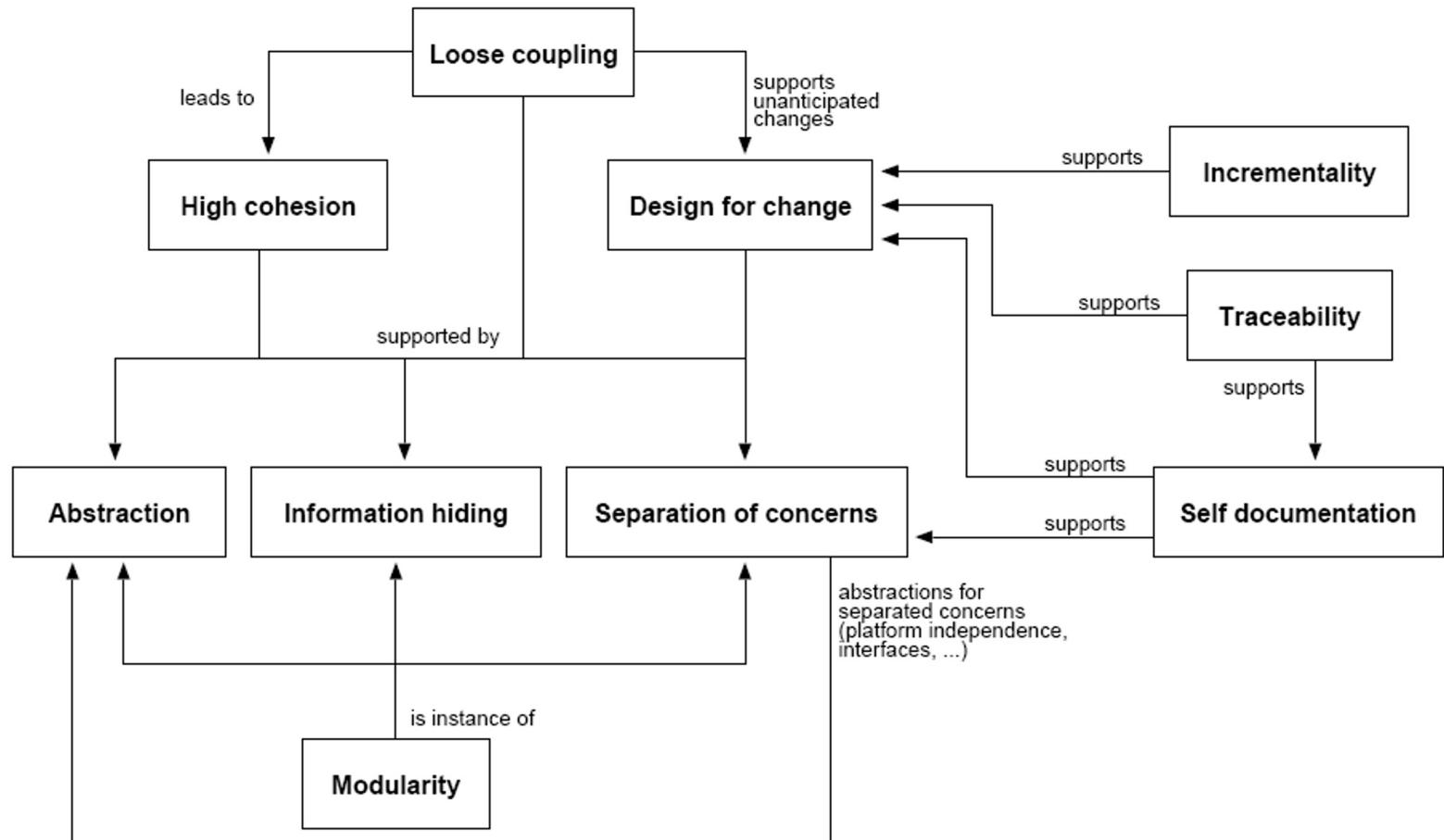
Introduction and adherence to standards, guidelines, principles, etc.

## Ultimate challenge

“to architect and design timeless and (ultra) large-scale software...”  
(not only buildings, but cities of buildings...)

Brown, Simon. Software architecture for developers. Leanpub, 2015.

# Software Architecture: key principles



# Software Quality Attributes

Architectural **evolvability** is about preserving a system's software qualities over time

- |                    |                    |                        |
|--------------------|--------------------|------------------------|
| ❑ Accessibility    | ❑ Configurability  | ❑ Effectiveness        |
| ❑ Accountability   | ❑ Correctness      | ❑ Efficiency           |
| ❑ Accuracy         | ❑ Credibility      | ❑ Extensibility        |
| ❑ Adaptability     | ❑ Customizability  | ❑ Failure-transparency |
| ❑ Administrability | ❑ Debuggability    | ❑ Fault-tolerance      |
| ❑ Affordability    | ❑ Determinability  | ❑ Fidelity             |
| ❑ Agility          | ❑ Demonstrability  | ❑ Flexibility          |
| ❑ Auditability     | ❑ Dependability    | ❑ Inspectability       |
| ❑ Autonomy         | ❑ Deployability    | ❑ Installability       |
| ❑ Availability     | ❑ Discoverability  | ❑ Integrity            |
| ❑ Compatibility    | ❑ Distributability | ❑ Interchangeability   |
| ❑ Composability    | ❑ Durability       | ❑ ...                  |

# Reusing architectural knowledge

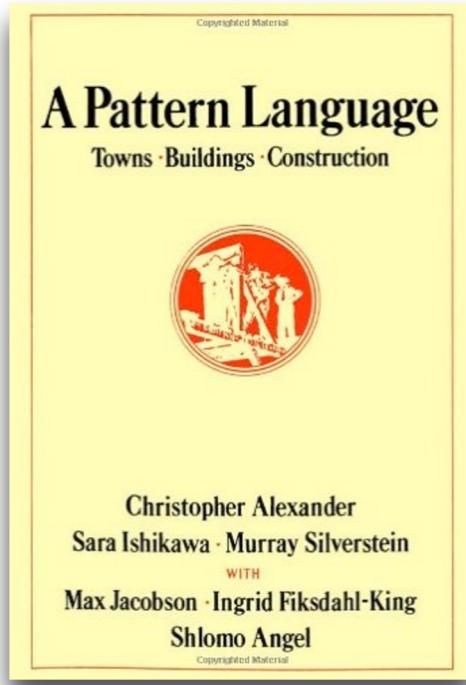
To architect solutions from scratch is hard and not effective in most cases

Good architects **reuse** solutions that worked well before and adapt them smoothly and incrementally to new situations.

Therefore it is very important:

- to know well proven solutions
- to reuse generic solutions
- to adapt existing solutions
- and then to add innovation over them to fit the needs.

# Architectural Patterns, 1977



Christopher Alexander in this book presents a pattern language, an ordered collection of 253 patterns.

The goal was to enable non-experts to architect and design their own houses and communities.

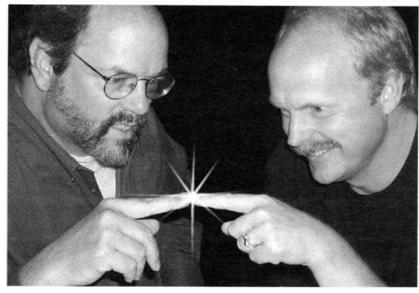
Patterns describe current practice and provide vision for the future.

Patterns integrate knowledge from diverse sources and link theory and practice.

Patterns have a strong “bottom up” orientation.

<http://c2.com/cgi/wiki?ChristopherAlexander>

# Software Patterns, 1987..1994..



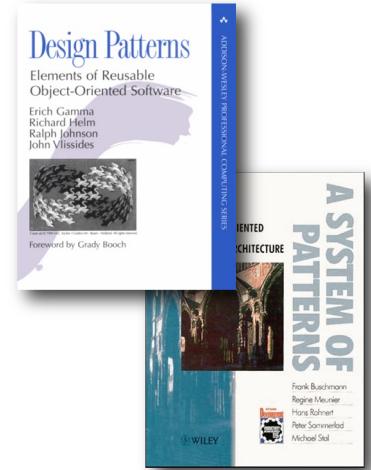
Kent Beck & Ward Cunningham,  
“Using Pattern Languages for  
Object-  
Oriented Program”, OOPSLA ‘87,  
1987.



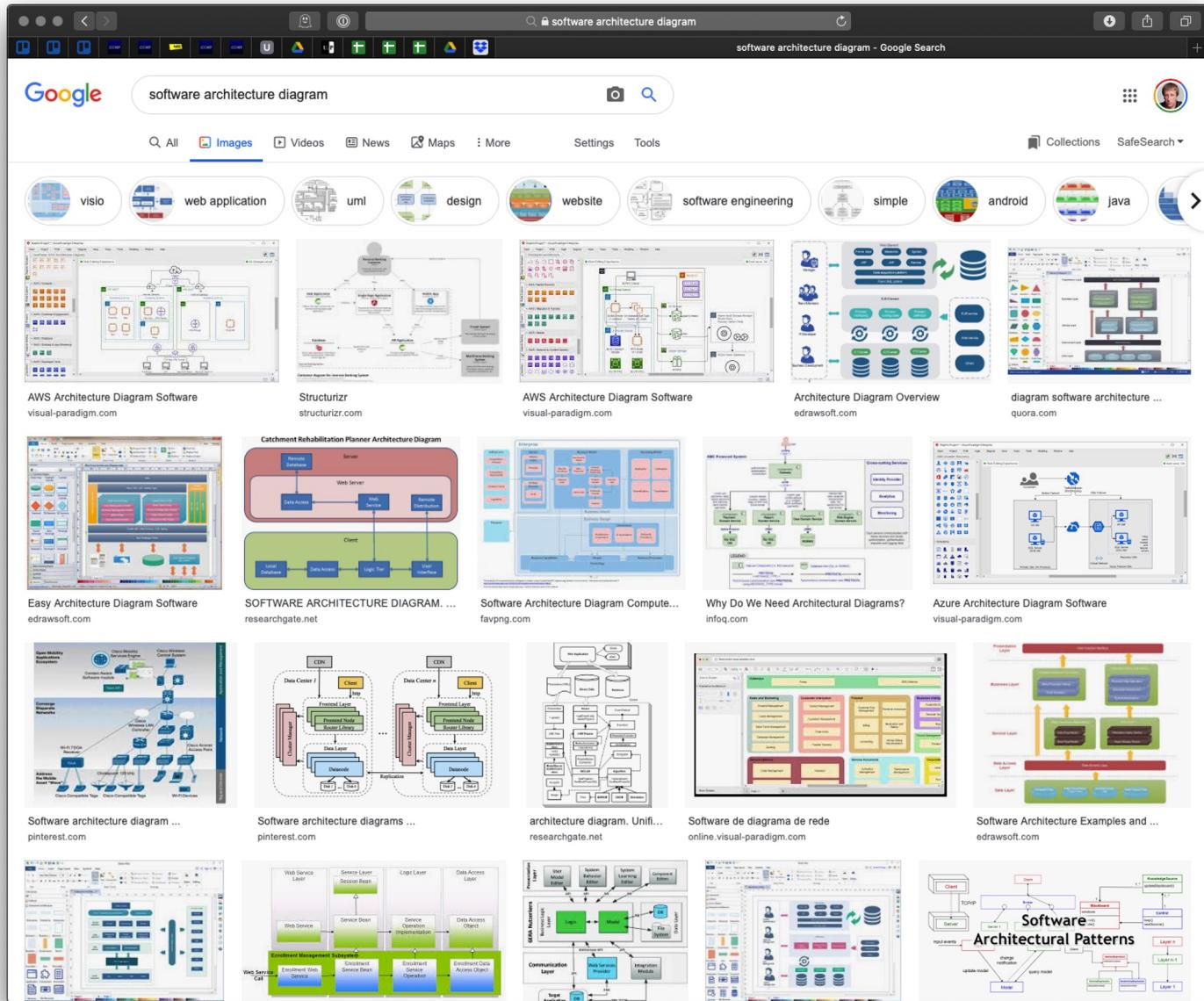
A new software patterns discipline started by a group later called “The Hillside Group”, 1993.



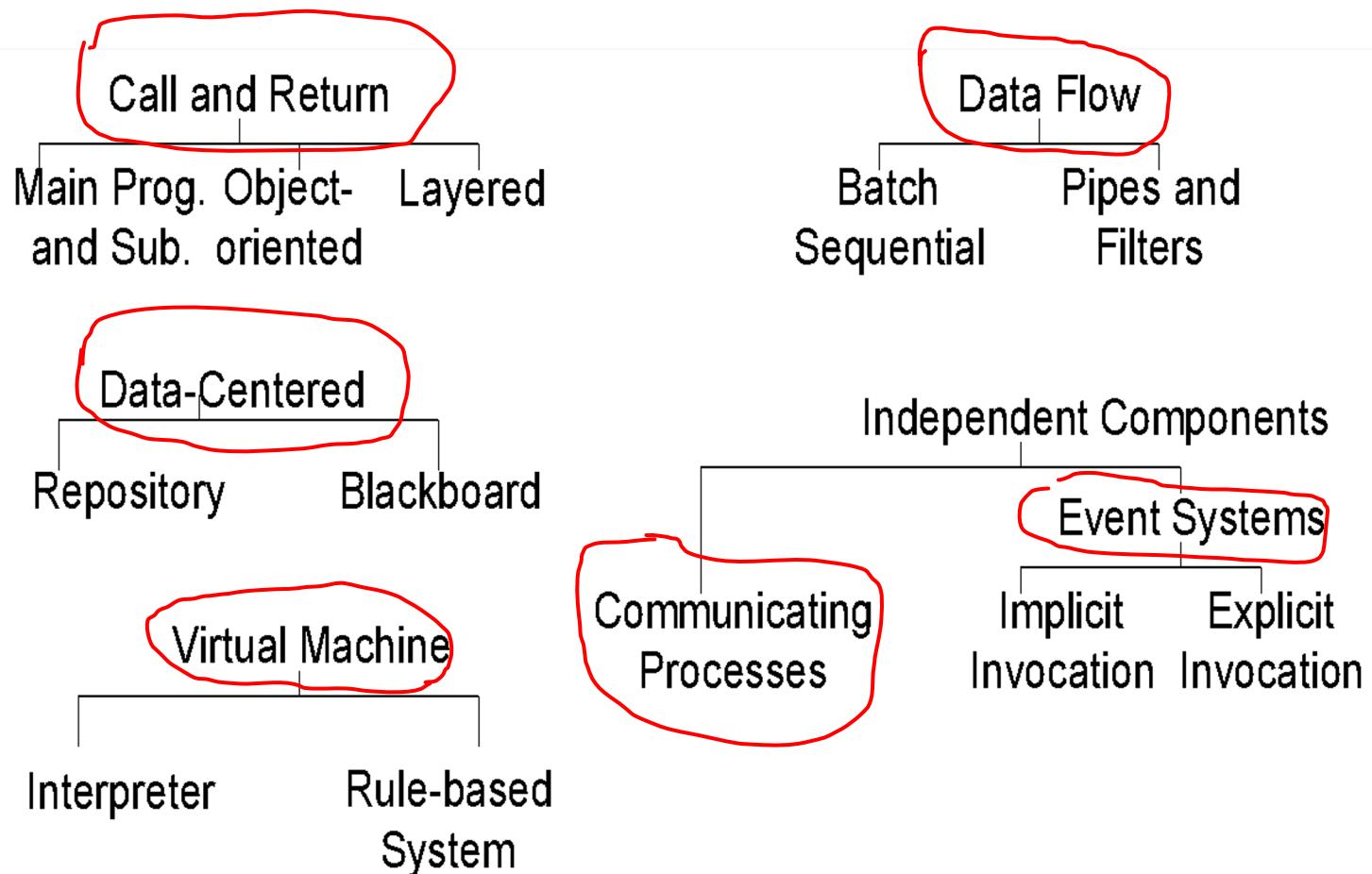
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (aka Gang of Four), published the first book on “Design Patterns”, the “bible”, 1994.



# Software Architecture: communicating



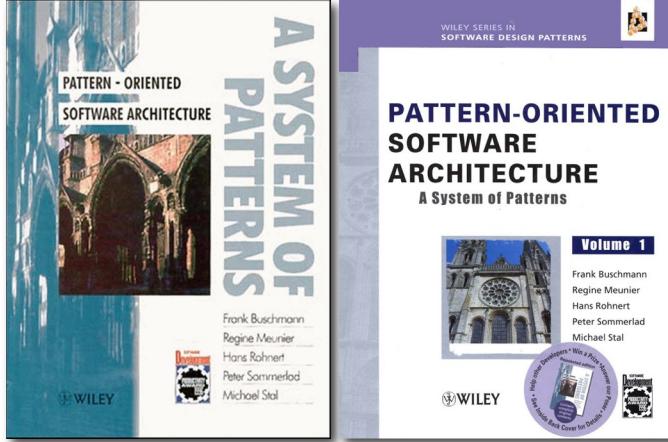
# Architectural styles



*From Chapter 5, Software Architecture in Practice, p. 95*

# **POSA Patterns**

## **“Pattern Oriented Software Architecture”**



**Pattern Oriented Software Architecture,  
A System of Patterns**

Frank Buschmann  
Regine Meunier  
Hans Rohnert  
Peter Sommerlad  
Michael Stal

**Wiley, 1996**

# Cloud, Microservices & DevOps Patterns

## Cloud Adoption Patterns

Patterns for Developers and Architects building for the cloud

- Cloud Adoption
- Cloud Client Architecture
- Microservices Design
- Microservices
- Strangler Patterns
- Event-based Architecture
- Coexistence Patterns
- Cloud Native DevOps
- Scalable Store
- Container DevOps
- Cloud Security Patterns
- Organization and Process Patterns
- Container Building

Search Cloud Adoption Patterns

## Patterns for Developers and Architects building for the cloud

```
graph TD; CA((Cloud Adoption)) -- Often requires --> CCA((Cloud Client Architecture)); CA -- requires --> CND((Cloud Native DevOps)); CCA -- Depends on --> SP((Strangler Patterns)); CCA -- Implemented through --> EBA((Event Based Architecture)); CCA -- often based on --> CP((Coexistence Patterns)); CND -- Built using --> M((Microservices)); M -- Leads to --> CND; M -- requires for persistence --> SD((Scalable Store)); M -- Designed and deployed using --> CNA((Cloud Native Architecture)); M -- facilitates --> CP; SD -- often implemented using --> CD((Container DevOps));
```

The diagram illustrates the relationships between various cloud patterns, centered around Microservices. Key relationships include:

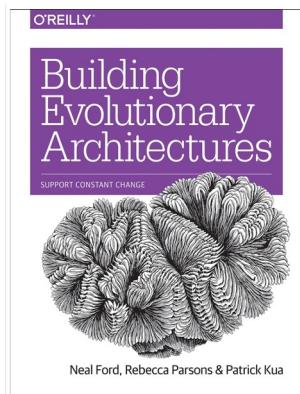
- Cloud Adoption** often requires **Cloud Client Architecture**.
- Cloud Adoption** requires **Cloud Native DevOps**.
- Cloud Client Architecture** depends on **Strangler Patterns** and is implemented through **Event Based Architecture**.
- Cloud Client Architecture** is often based on **Coexistence Patterns**.
- Cloud Native DevOps** is built using **Microservices**.
- Microservices** leads to **Cloud Native DevOps**.
- Microservices** requires persistence from **Scalable Store**.
- Microservices** is designed and deployed using **Cloud Native Architecture**.
- Microservices** facilitates **Coexistence Patterns**.
- Scalable Store** is often implemented using **Container DevOps**.

<https://kgb1001001.github.io/cloudadoptionpatterns/>

# Evolutionary architecture

*How is long-term planning possible when things are constantly changing in unexpected ways?*

*How can we build architectures that can gracefully change over time?*



**Building Evolutionary Architectures**

Neal Ford  
Rebecca Parsons  
Patrick Kua

O'Reilly, 2017

# UC Components

## TP classes

- 14 weeks x 3 hours (1,5 T + 1,5 P) = 42h
- Lectures
- Case studies
- Development of projects
- Q&A sessions

# Case studies & Projects

## Case studies

- from web, publicly available
- from companies, by invited speakers

## Architecture and Design Projects

- several small projects in groups (almost no coding)
- practice software design and architecture
- presentations to each other about their designs
- application of patterns

# Grades

Distributed evaluation, no final exam, 1 mini-test.

## Components

- 1 mini-test with access to materials, 90min, before May.
- 8-10 Assignments. Most are group projects that involve designing the architecture of a system.
- Individual assessment based on student's performance.

## Formula

- $(\text{Mini-test} \times 30\%) + (\text{Assignments} \times 60\%) + (\text{Individual} \times 10\%)$

# Self-formation of teams

Pick a team and add your GitHub username in the spreadsheet using your official Google account provided by the University.



# Bibliography

- [Alexander77] C. Alexander and S. Ishikawa and M. Silverstein, A Pattern Language, Oxford University Press, 1977.
- [Alexander79] C. Alexander, A Timeless Way of Building, Oxford University Press, 1979.
- [Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [Buschmann96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern Oriented Software Architecture - a System of Patterns, John Wiley and Sons, 1996.
- [Buschmann99] F. Buschmann, Building Software with Patterns, EuroPLoP'99 Proceedings.
- ...

# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**

# **Introduction: Software Architecture**

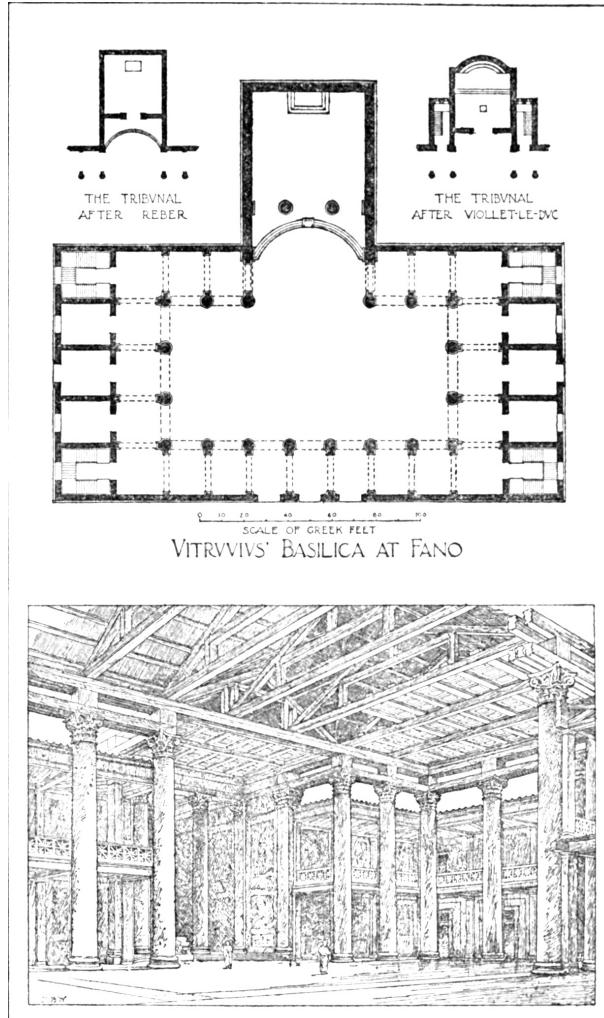
What is Software Architecture?

Why is it important?

What is Architecture (building, not software)?

What is an Architect (a building architect, not a software architect)?

# Marcus Vitruvius Pollio (b. 80-70 BC)



# Vitruvius: 3 Pillars of Architecture



**Utility**

**STABILITY**

**Beauty**

# **Field Trip!!!!!!!!!!!!!!**

Let's look for those three pillars of architecture:

- Firmness (stability)
- Utility (usefulness)
- Beauty

# Utility: Fit for the purpose

Interior of Notre Dame  
Cathedral, Paris

Encourages visitors to  
lift their eyes  
heavenward



# Firmness: Romanesque Architecture



# Firmness: Building a Cathedral

Flying buttresses,  
Reims (France)  
Cathedral

Enabled  
construction of:

- High thin walls
- Large windows



# Unity of Design and Appearance (Frauenkirche, Munich)



# **Unity of Design and Appearance (II)**

## **Aachen, Germany**



# Unity of Design and Appearance (III)

Denver Public Library, USA

Yes, this is ONE building!

Designed by a committee of architects!

- Could you guess?



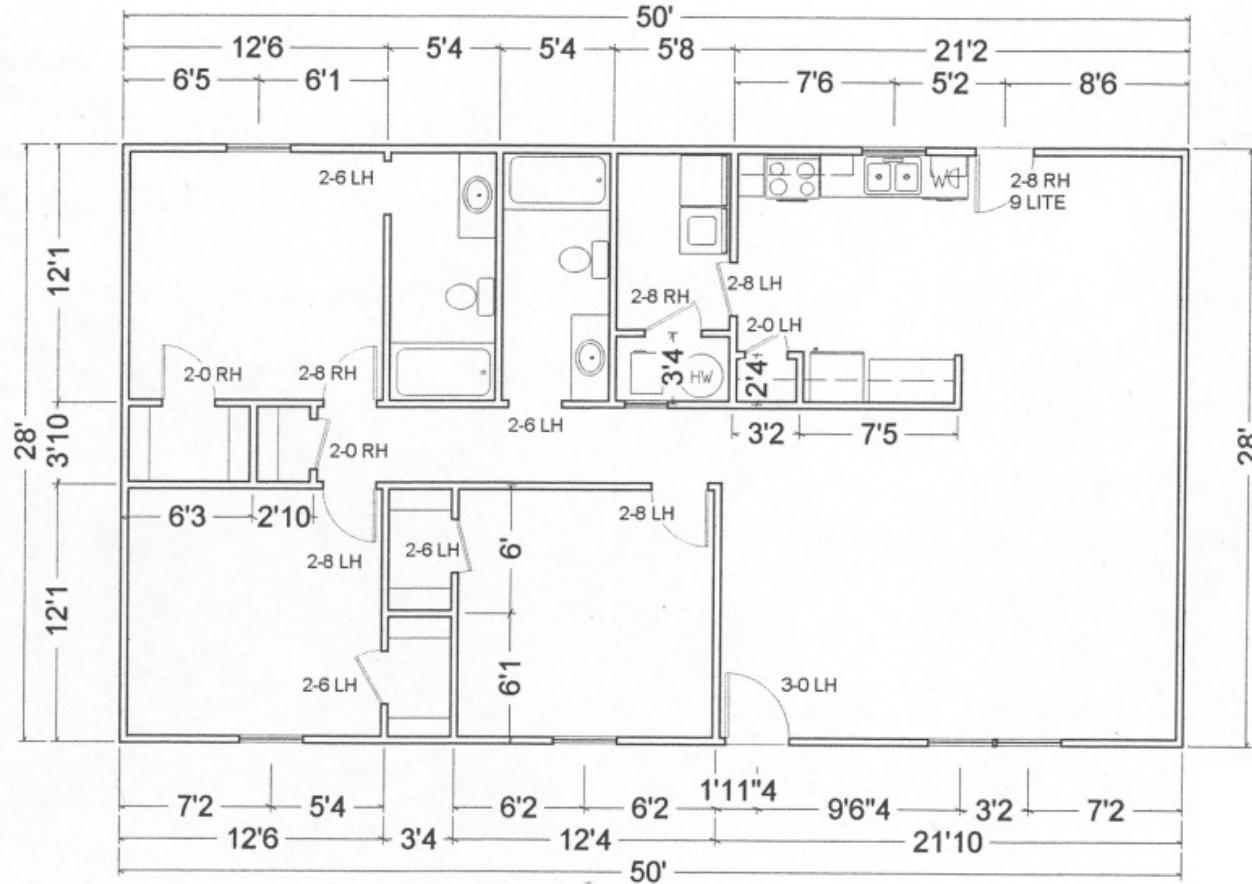
# Path for Evolution

Chartres Cathedral, France

Architectural drift!



# Partitioning



Also:

- Interfac
- Construction directions

# What is Software Architecture?

Well, we know what Building Architecture is:

- The Structure of the Building
- The Overall Partitioning – Rooms and Such
- Guidelines for Construction
- Some notion of the use of the building

It is similar in Software Architecture

- The gross structure of the software
- The overall partitioning – Subsystems and Such
- Guidelines for construction, e.g., technologies to be used
- The highest-level mapping of the problem space to the solution space
- Some notion of the use of the software

# ... more

A model of the finished system

- Related to mapping of the problem space to solution space

A vision

- Of how the system should work
- Of how it should be implemented

A guide for implementation

- (see above)

# Why Software Architecture?

The bits don't care!

Why is software architecture important?

# (one reason)

Think back on the cathedrals:

- How did architecture affect the end-user experience?

Software architecture:

- How can software architecture affect the end-user experience?
- Give examples...

# Why Architecture?

Every system has an architecture

- But some are better than others
- A systematic architecture makes life much easier for future as well as present developers

# Architecture helps:

People understand the system

- Programmers, managers, **and users**

Divide up the work

Accomplish quality requirements

- Important: more on this later

Maintenance and enhancements

Guide product families

# Architectural Knowledge

In system maintenance & enhancement, how much effort is devoted to discovery?

- HALF
- A good architecture can help a lot
  - Structure
  - Implementation conventions

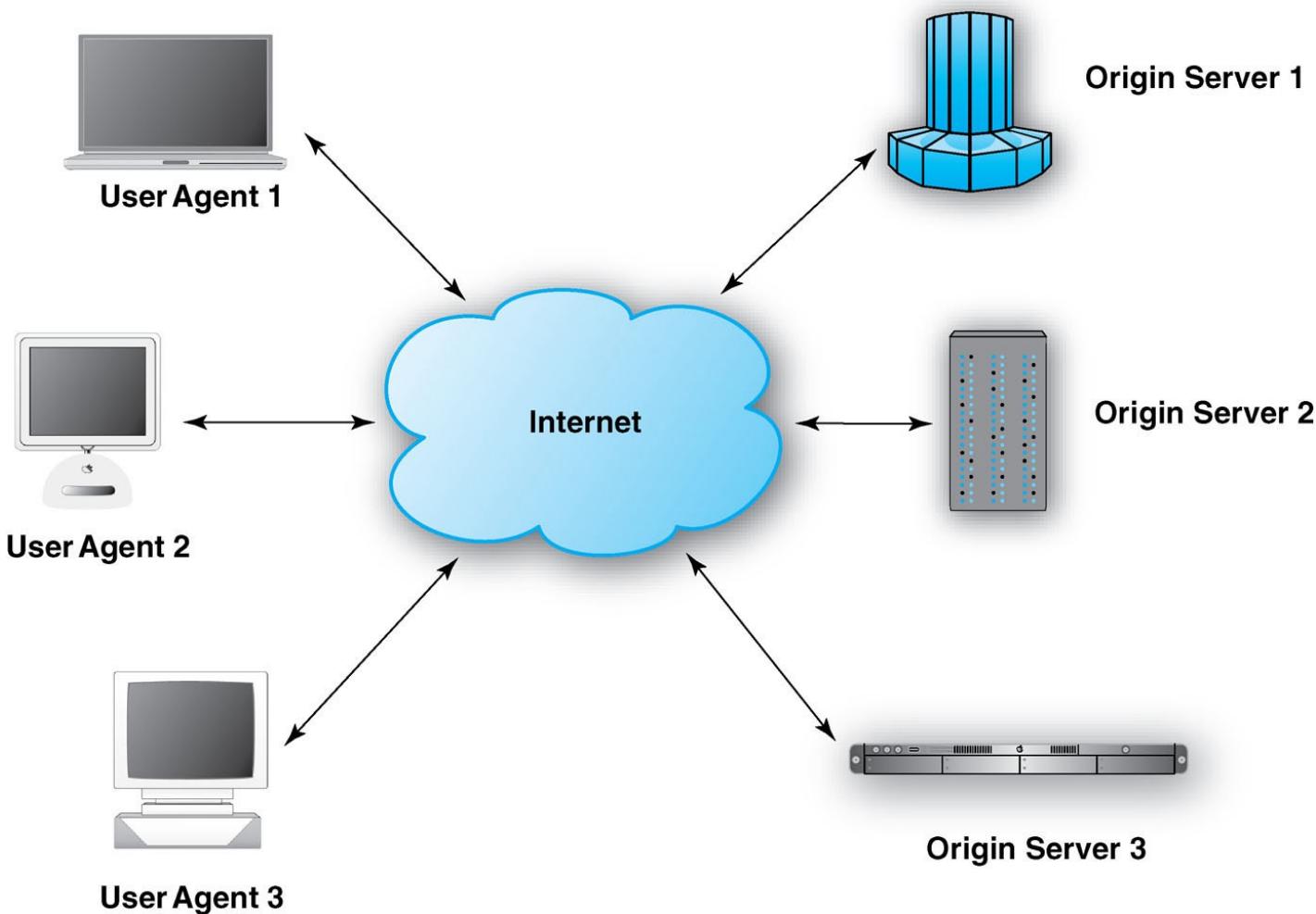
# The Power of Big Ideas: The Architecture of the Web

Suppose you want to build the World Wide Web ...

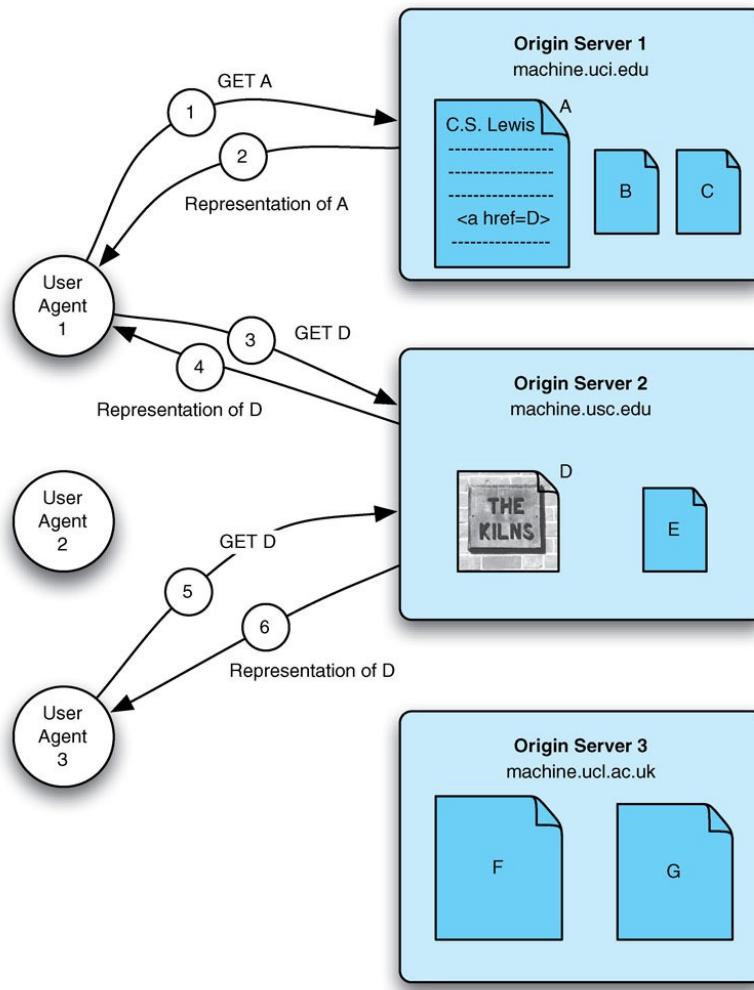
Where do you start?

For that matter, what *IS* the Web?

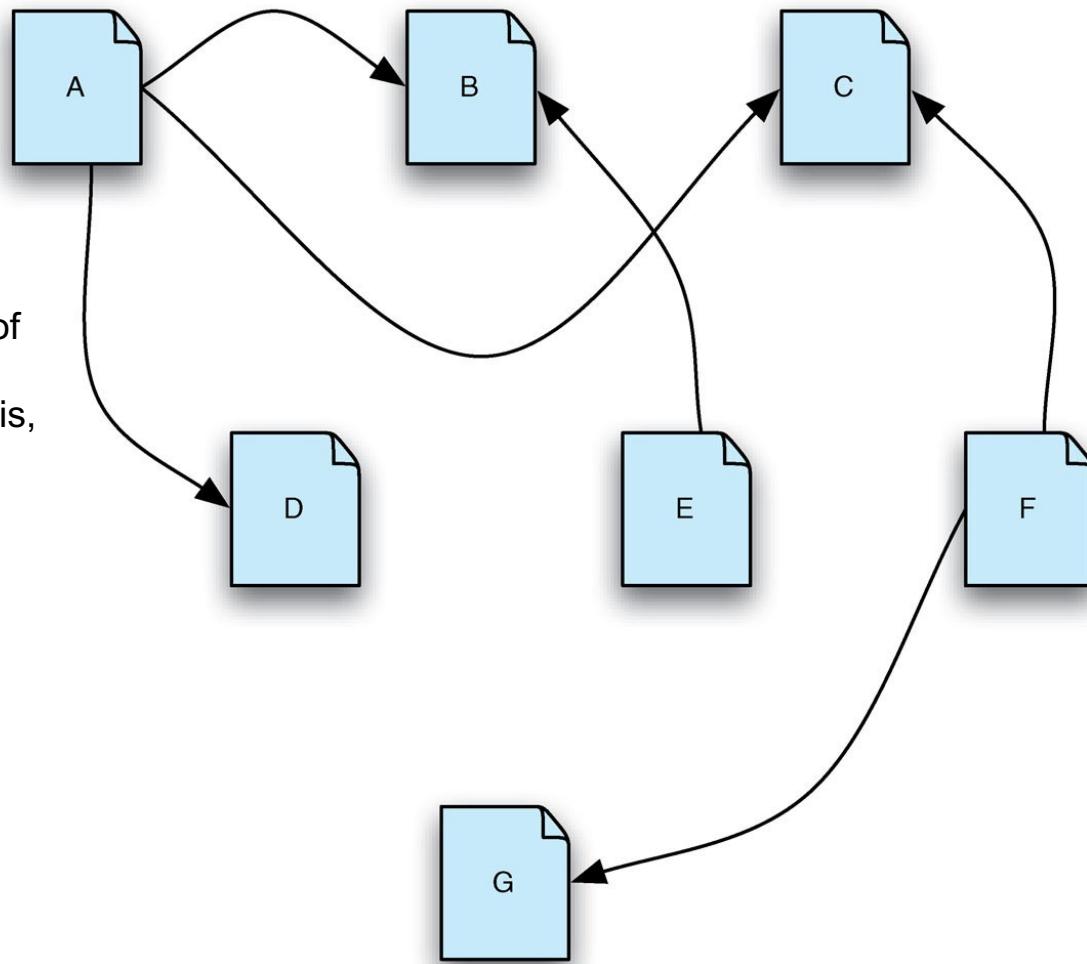
# One View of the Web



# Another view



# Yet Another View of the Web



A user might view it as a set of linked information:

- a. A biography of C. S. Lewis, an author
- b. One of his books
- c. Description of where he lived
- d. Picture of his house
- e. Etc.

# Another view of the Web!



# Web Architecture

What do we learn (about the Web's architecture) from these pictures?

What is missing that is important?

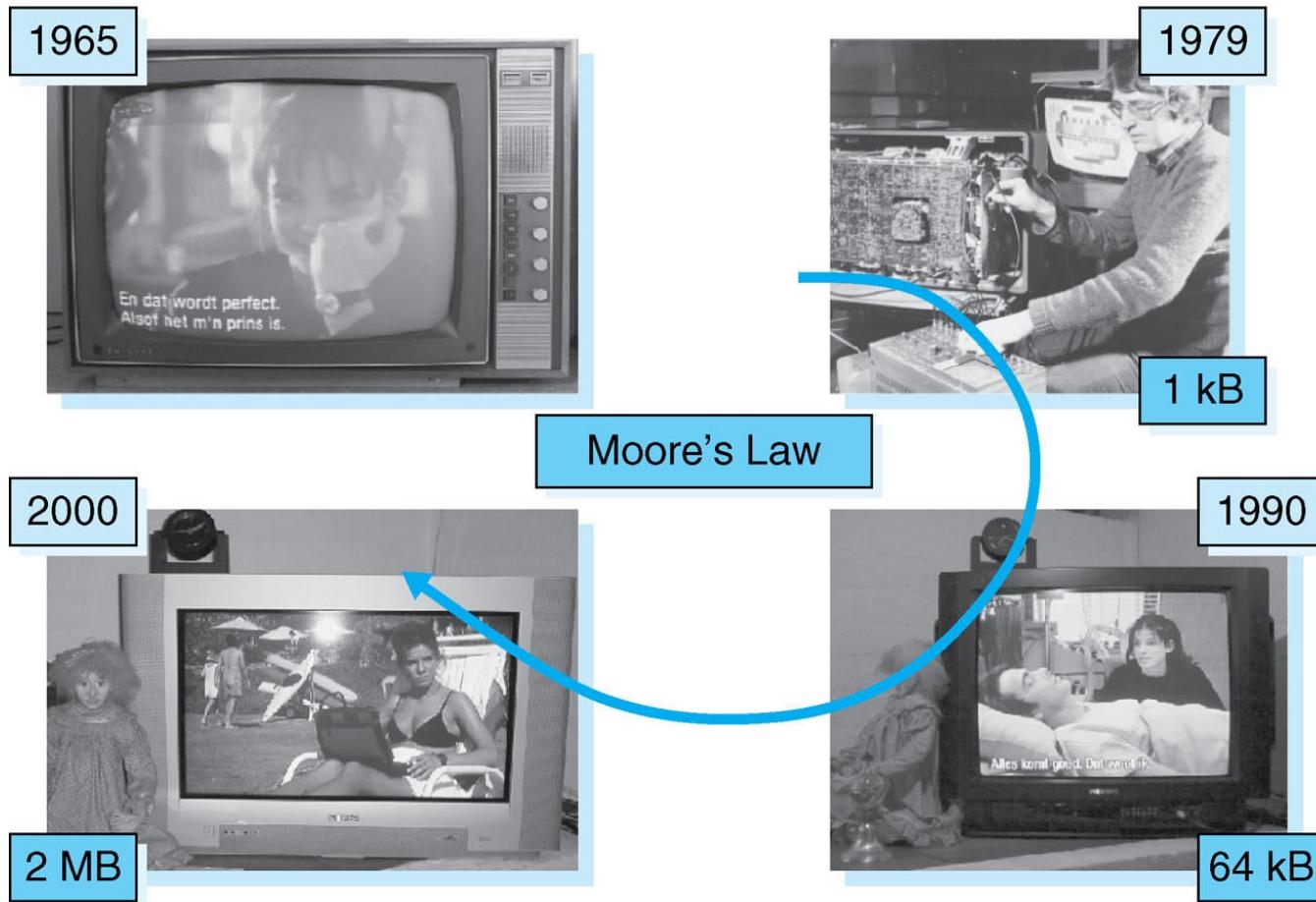
Why were the decisions about the architecture made?

Why these decisions and not others?

Why did similar systems fail, but the Web succeeded?

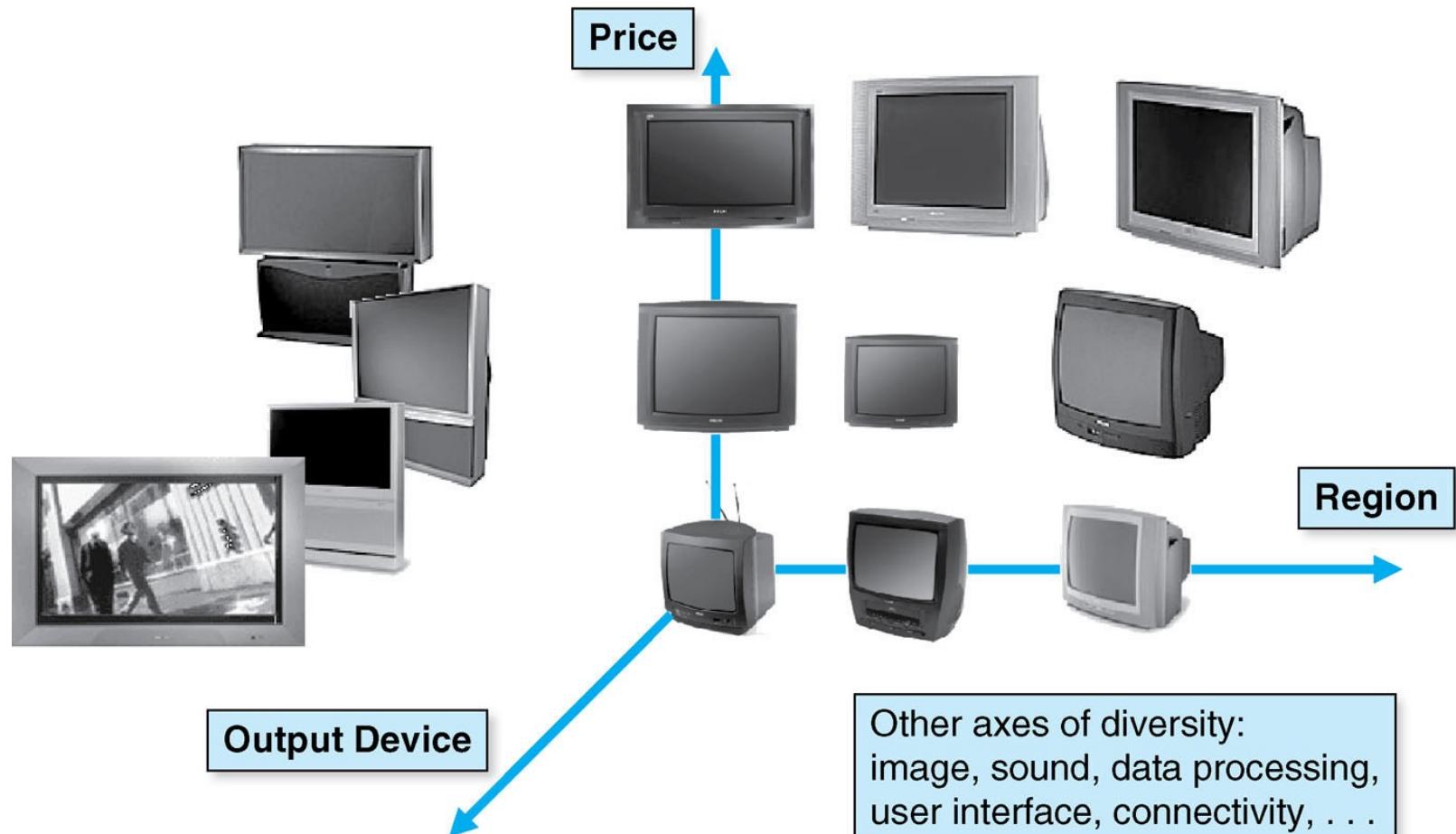
# The power of Architecture: Enabling product lines (Phillips)

## Complexity

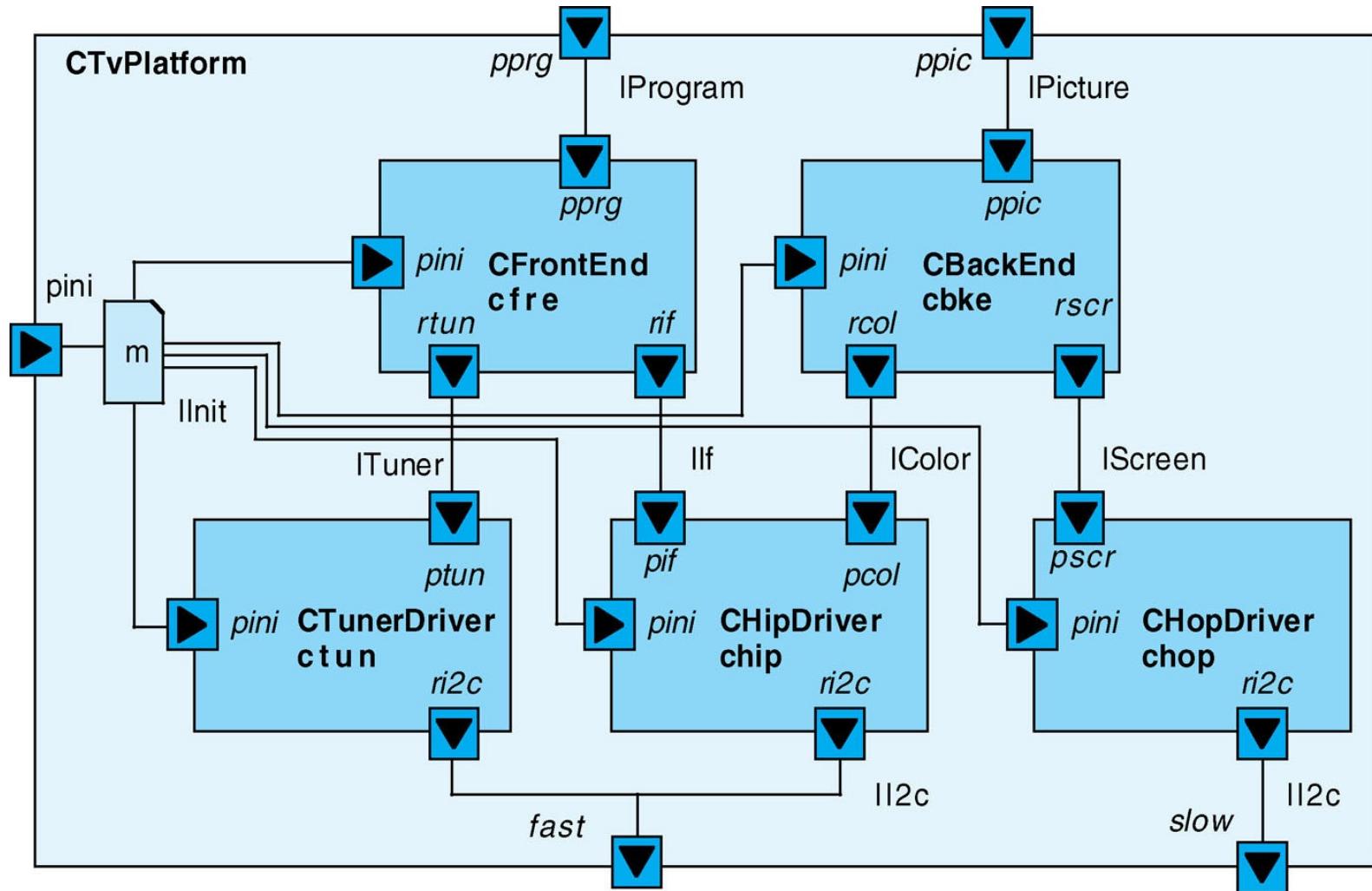


# Phillips Product Families

## A Television Product Family

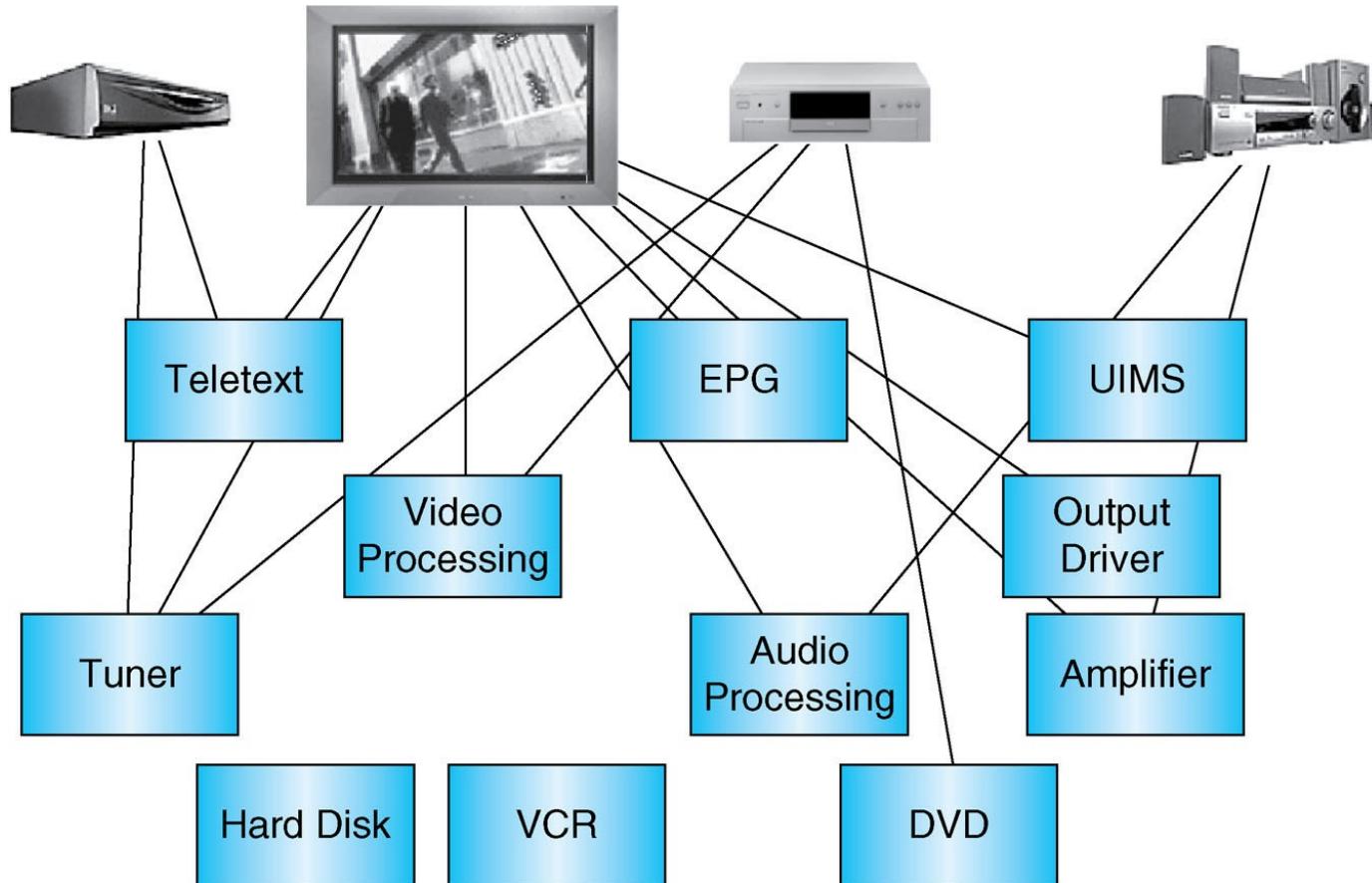


# TV Set Software Architecture



# Possible products

## Composition



# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**

# **Architecture, Ambiguity, and Abstraction**

# Let's talk about Specifications ...

Let's play fantasy developer!

You are on a team writing a new payroll management system for the company.

In small groups, discuss

- In an ideal world, what does the spec look like?
  - General characteristics
  - Give several examples
  - (write some notes)

# If you are the architect ...

What does the spec look like (real world)?

**“We need a new payroll management system”**

Why?

It's the nature of architecture: very little has been decided yet.

**If you are the architect ...  
and you want a detailed spec:**



# The nature of architecture is ambiguity

Practical ramifications for this class:

- Assignments are ambiguous
- Your solutions will vary (from each other)
- Your solutions will vary from mine!
- Grading can be ambiguous

Don't let it rattle you!

# Quick Quiz

Developers produce code. Who is the consumer of the code?

The Compiler!

- Your code HAS to meet the demands of the Compiler
- Compilers are NOT ambiguous!

Did you say “end user”?

- Well, it's really nice if it satisfies the end user.
- Or at least it satisfies people who buy it (they may or may not be the end user.)

Architects produce architectures. Who is the consumer?

- PEOPLE! (including, but not limited to developers)
- And people are ambiguous!

# Understanding is in the eye (and ear) of the beholder



I know you think you understand  
what you thought I said, but I'm not  
sure you realize that what you heard  
is not what I meant.

— Robert McCloskey —

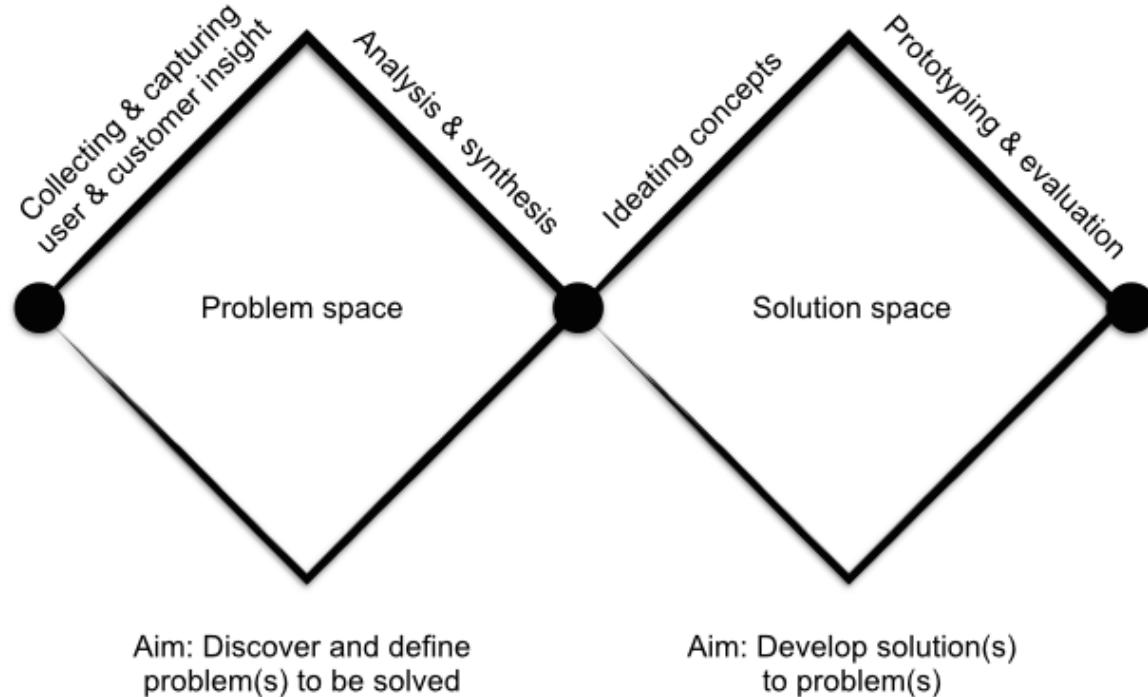
AZ QUOTES

# Problem Space

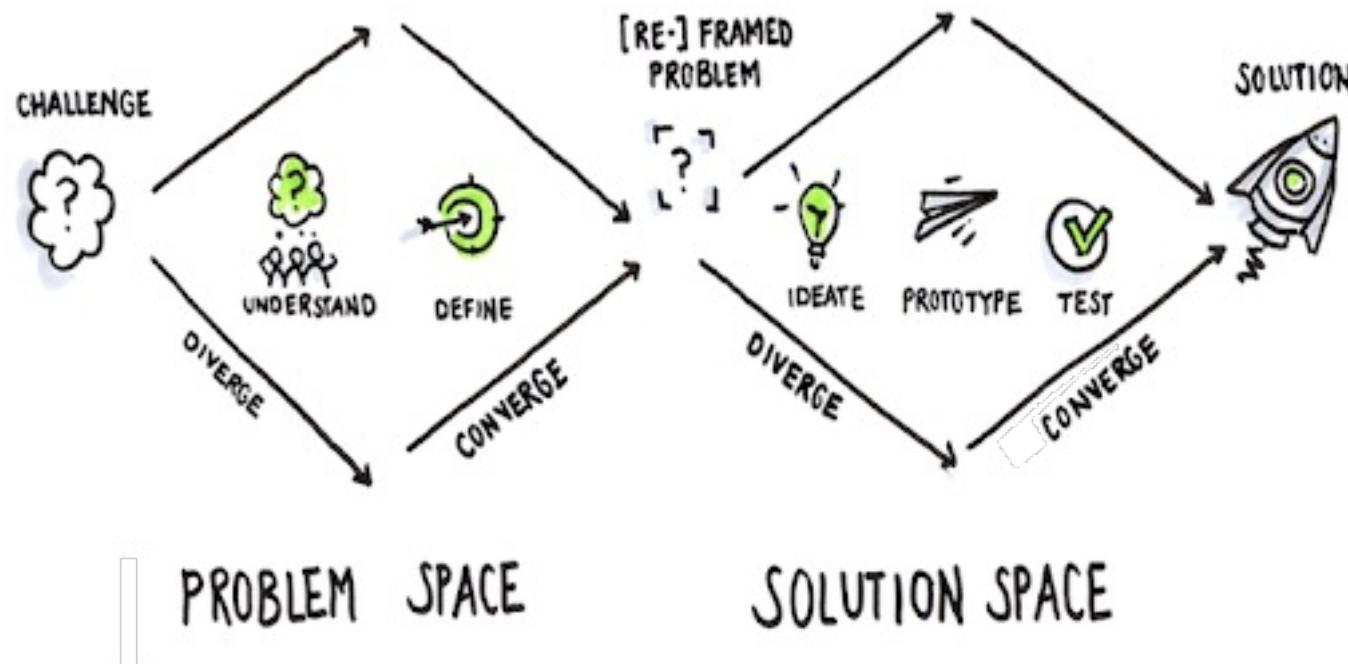
# Solution Space



# Activities in each:



# Place the architect in this diagram



# Example #1

## Problem Space vs. Solution Space

### ■ Problem Space

- A customer problem, need, or benefit that the product should address
- A product requirement

Example:

- Ability to write in space (zero gravity)

### ■ Solution Space

- A specific implementation to address the need or product requirement



- NASA: space pen (\$1 M R&D cost)

- Russians: pencil



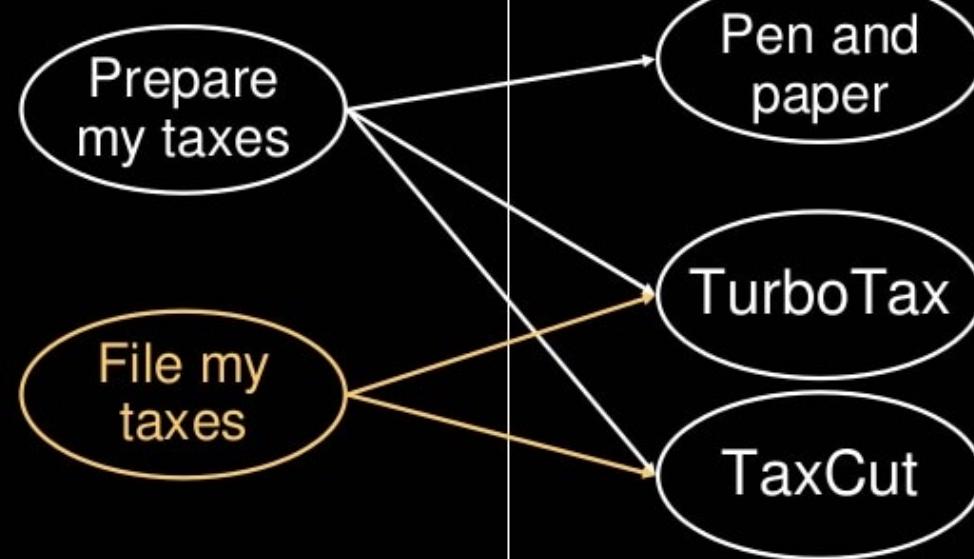
Copyright © 2010 YourVersion

## Example #2 (product level)

### Problem Space vs. Solution Space Product Level

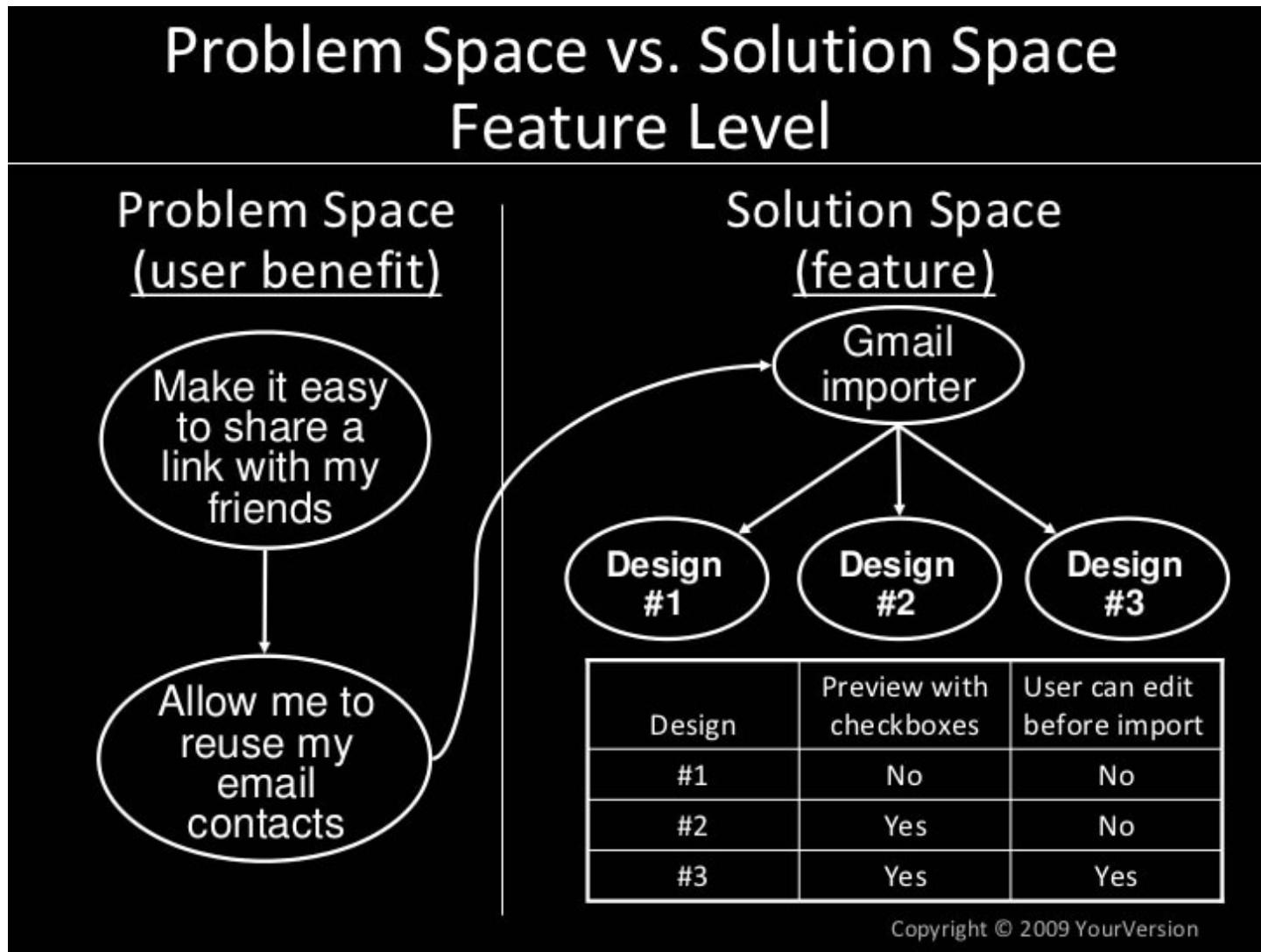
Problem Space  
(user benefit)

Solution Space  
(product)



Copyright © 2009 YourVersion

# Example #3 (feature/architecture level)



# A few words about roles

What role(s) usually work in the problem space?

Analysts, spec writers, surrogate users, Product Owners, etc.

**And often architects!**

What role(s) usually work in the solution space?

**Architects**, (other) designers, implementers, programmers, etc.

NOTE: in some companies, the architect does it all!

# Not Waterfall

Early in the project:

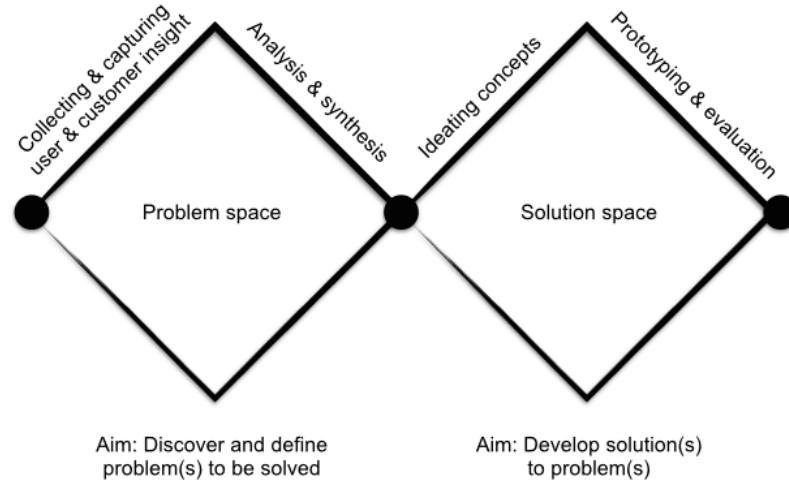
Note that work in both diamonds proceeds simultaneously.

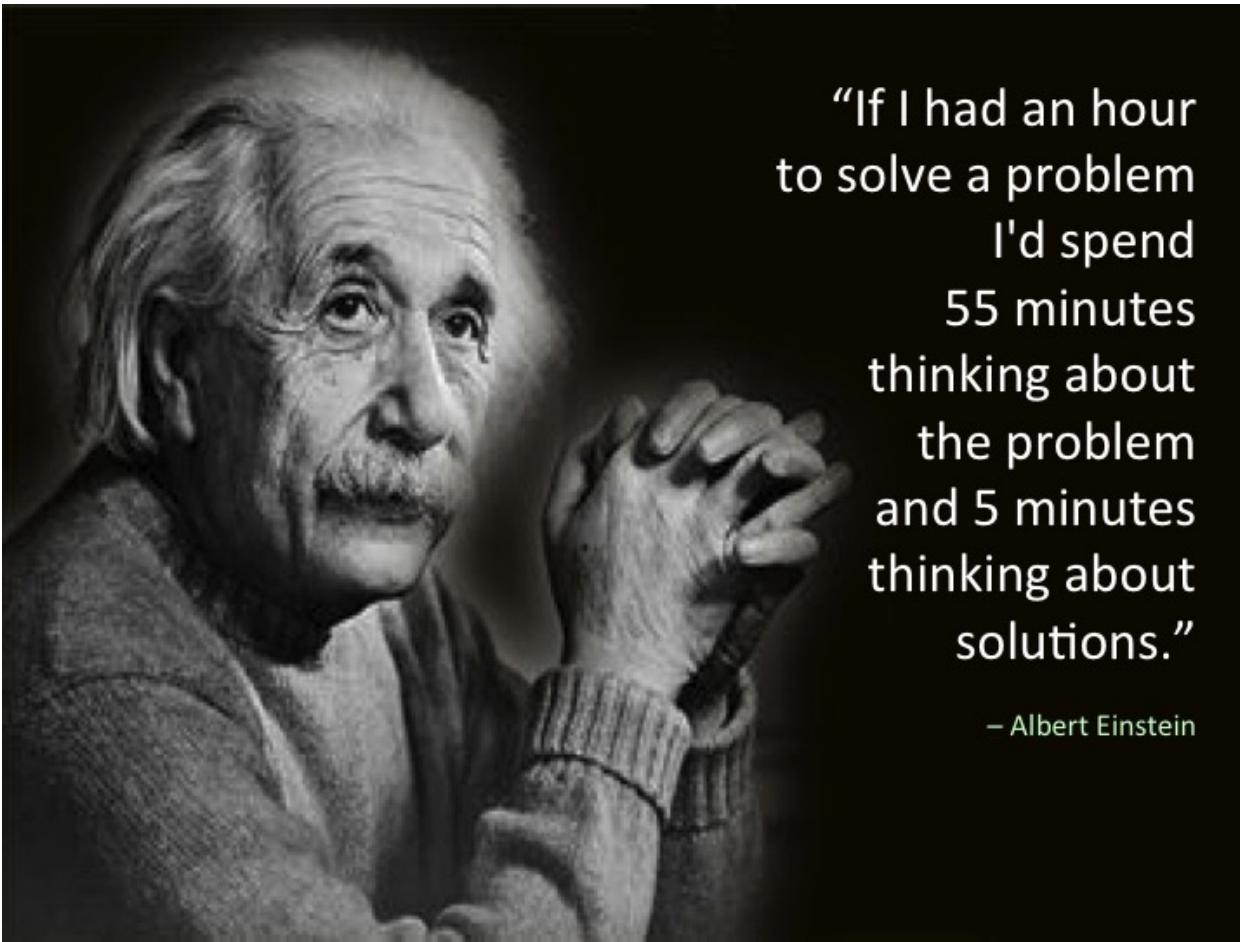
While you as architect are designing the architecture:

- The requirements are being analyzed

The things YOU learn are input to the problem space too!

- Can you think of examples?





“If I had an hour  
to solve a problem  
I'd spend  
55 minutes  
thinking about  
the problem  
and 5 minutes  
thinking about  
solutions.”

– Albert Einstein

# Architecture Exercise

You are a lead developer in a company. The company makes software for medical billing: it manages billing of medical procedures first to insurance companies, and then bills remaining fees to customers. This is a very sophisticated system, as it must deal with multiple insurance companies (presumably each has its own API), as well as customers. And it should interface nicely with whatever software the doctor uses for the medical records and exam/treatment records.

One day your boss comes in and says, “I just got out of a meeting with the company executives. They see that there is a market opportunity for an app associated with our product. Our group will develop it. I’m appointing you to be the architect. Congratulations!”

# What do you do?

1. Run screaming from the room
2. Accept, with some concerns
3. Call up your friend who is an app developer and say, “What’s an app?”
4. Start coding right away, and hope you can get a shell demo done by next week

Ok, you agreed to be the architect for the app. You no doubt have some questions.

Write the first question you will ask:

- (About the app itself, NOT about the schedule)

(only one question right now)

# Attempt #1

Was your question: “Apple or Android”?

If so, 20 points from Slytherin!

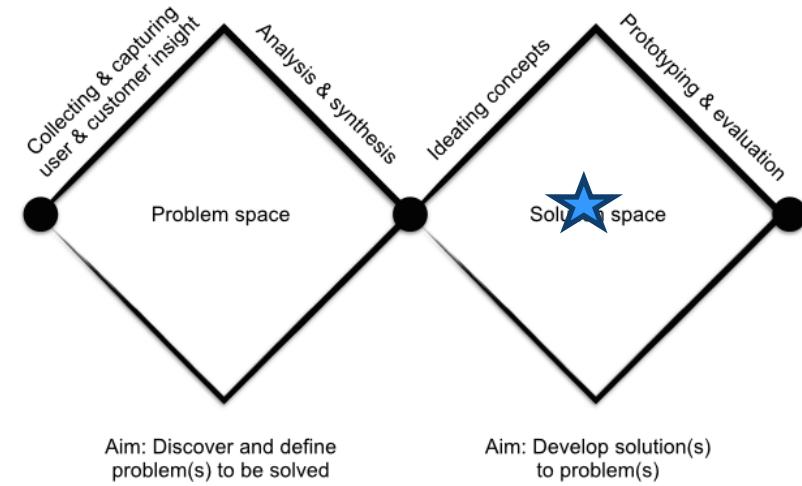
Why?

Because:

- This is way too early to ask this question.
- It IS an architectural question, but its focus is on implementation. Right now, you need more information on the other half: on the problem space rather than the solution space.
- So you have the wrong mindset.

# Apple or Android?

Place this question in  
the diagram:



# Attempt #2

Was your question: “Please give me a feature list for the App?”

If so, 10 points from Gryffindor!

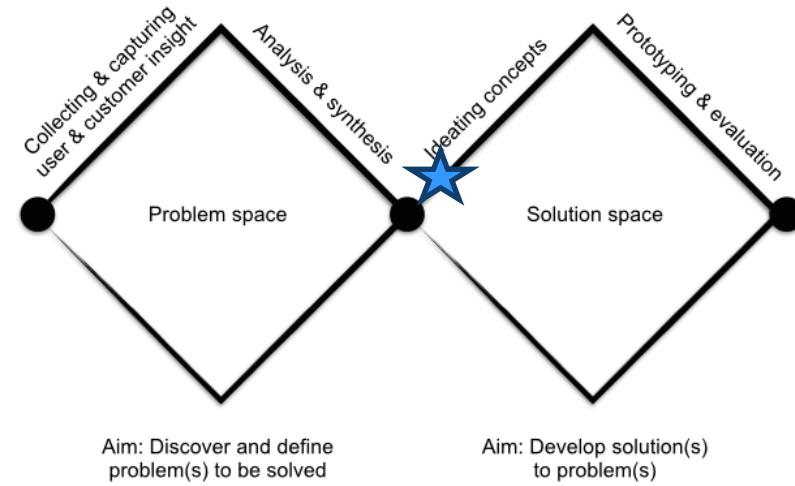
It’s pretty good. But what’s wrong with this question?

If you don’t know who will use it, you can’t know what it should do. Otherwise, it’s a solution looking for a problem.

This is also necessary for the architecture, but first things first!

# What is the list of features?

Place this question in  
the diagram:



# Attempt #3

Was your question: “Who will use the app?” or “Who is the app for?” (or similar)

Great! 20 points for Ravenclaw!

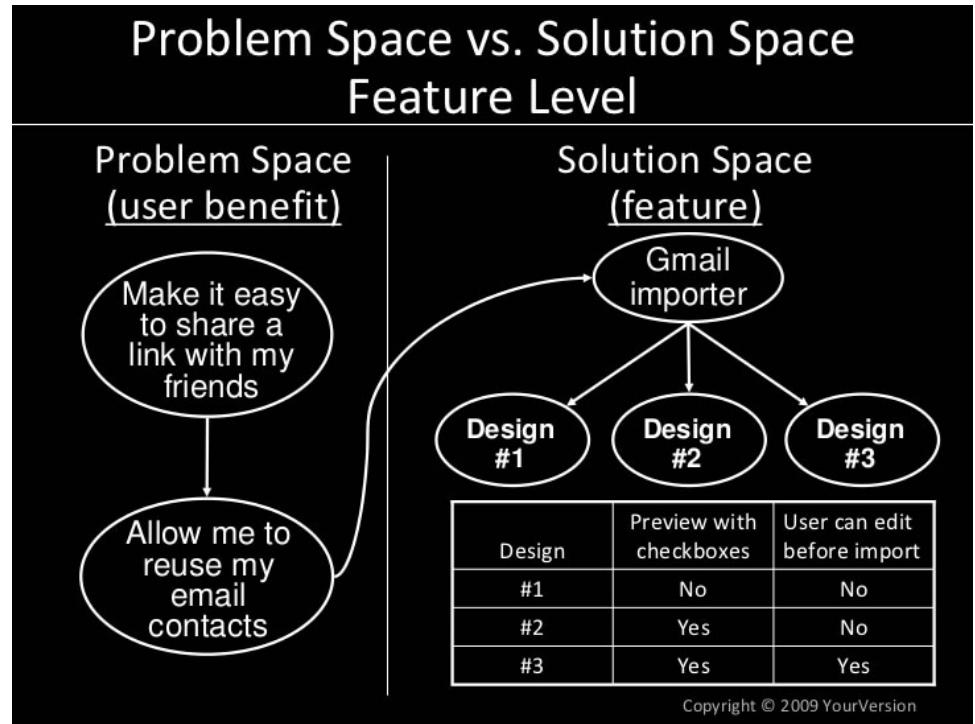
Why is this the starting point?

What similar questions will you want to also ask?

Ideas:

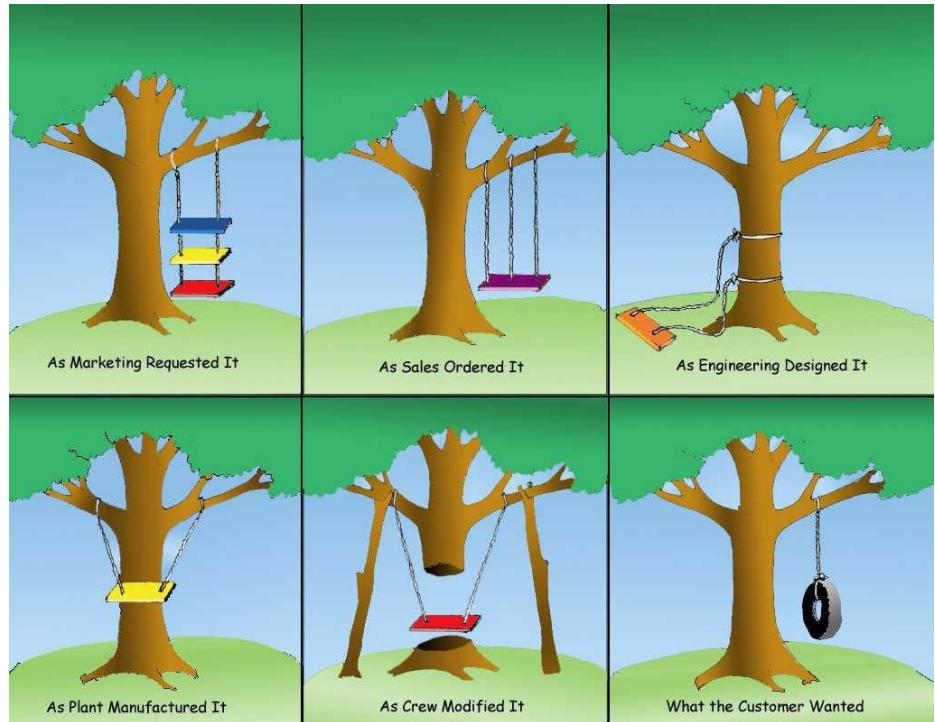
- What problem does the user have?
- What benefit will it be for the user of the app?
- Can they get that benefit anywhere else? (For example, suppose the app shows how much they currently owe. They might be able to get similar information from their insurance app. Or they can get through our company’s website, without using an app.)
- What is the compelling case for the app?

**(Reminder:  
features are in the  
solution space)**



# Problem and Solution Space

As architects, we must understand the problem space



# Analysis and Architecture

Analysis (of the problem space): figuring out what the user's problem is that we will solve in the product.

It's “requirements engineering”

That's not architecture!

But that looks like what you just did!

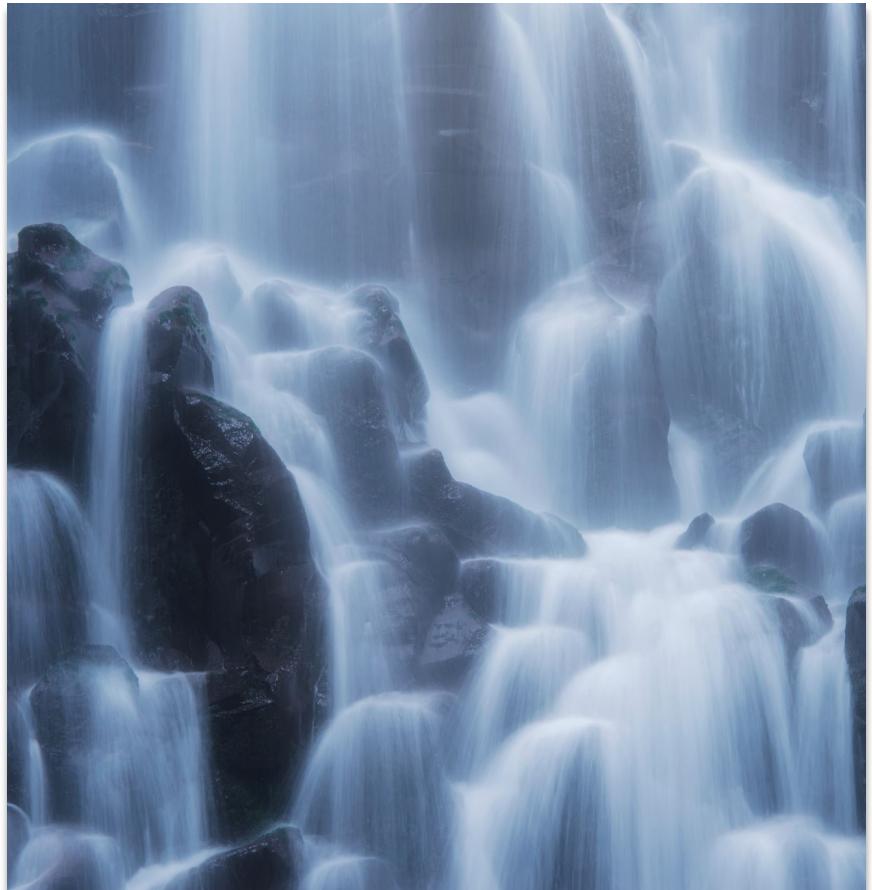
Since architecture is about a solution that solves the problem, architecture involves getting sufficient clarity of the problem space to create the solution

# The Tip of the Waterfall

So why don't the analysts do their job, and get the requirements sufficiently specified?

Because they can't. At the very least, they need information from the architects

- Wait, isn't that backwards?
- No. (Think prototypes)



# **Analysts vs. Architects**

## **- another perspective**

As an architect, you are asking questions about the nature of the problem, so you can understand the problem

- And then design a solution to the problem

### **Making sure you are solving the right problem**

As an analyst, you dig up the answers to the questions

- Of course, you get to ask questions too!

Tangent (thought question):

- Where does marketing fit in this picture?

# Just the Starting Point

You will ask LOTS more questions during architecture

Early on, they will be mostly in the problem space

Later, they will be more in the solution space

You will bounce between problem space and solution space

You want to learn a lot about the problem space: in particular, anything that will affect the architecture of the product.

Sometimes, “I don’t know” is the answer you will get.

Example: “Apple or Android” will come up, but just not the first thing



## In-class exercise

Let's brainstorm all the questions an architect might ask about the proposed app

Here's the "spec" (the general idea) again:

An app for medical billing, related to the existing product: it manages billing of medical procedures first to insurance companies, and then bills remaining fees to customers. It interfaces with insurance companies, and with doctors' patient info software systems.

(note: some questions will depend on answers to other questions.)

# **The architect as a Bridge**

But Remember!!!

The Architect  
**MUST** also know  
about the solution  
space:

You must know how  
to implement!!!



# Reality!

If you design the architecture, can it actually be built?

Can you think of things in software that cannot be implemented?



# **Abstraction**

“Applied Ambiguity”

# Abstract Art

- ▶ What is this picture about?
- ▶ In 5 Minutes:
- ▶ Write down everything you can about this picture
- ▶ (Individual assignment)
- ▶ (artist: Pablo Picasso)



# Abstract Art

What did you write down about the picture?

What were you able to infer from the picture, even though it is rather abstract?

Why?

From this experience, what is abstraction, and what can it do?

# Abstraction

Abstraction is:

- The structured removal of information (you remove the details)
- Or: ignoring (not considering) details of something

OO Design

- We have base classes that are abstract
- It's more than that they can't be instantiated
- It's that they represent a family of related classes
- \* And we can ***think*** about them in terms of the abstract class

But there are layers of abstraction above the OO Design

- Architecture

# Architecture and Abstraction

Most architectural decisions are in the abstract area

We aren't worried about implementation details yet  
Implementation should not impact the architecture

**Problem:** but sometimes implementation details do affect the architecture

- But often it's that we just didn't consider it back at architecture time

# **Goldilocks: not too little, not too much**

A challenge in architecture:

- Too little abstraction (too much detail), and you get lost in the weeds
- Too much abstraction, and you don't have enough substance to work with.

The abstraction level changes with time

- Detail increases over time

But it's uneven: some things become detailed soon, others can be deferred until very late

Throughout the process, you always need to be able to find the abstract view of the system (the architecture; the theory)

# Spikes of Detail

Occasionally, we have to go into considerable detail

- Because the outcome might affect the architecture
- Or because it's how to clarify requirements
- Or because we need to investigate feasibility
  - Example: feasibility of a new technology

Example:

- How fast does the automobile detection system need to be?
  - (What if there is heavy, but fast traffic?)
  - (Can our hardware keep up)
  - (What capacity computers will we need? Anything special?)
  - So we might do some traffic modeling and prototyping

# Homework

It's all about coming up with questions.

See homework description

# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**

# Homework Discussion

Present your “top five” questions:

1. WHY is this question important?
2. WHOM does this question impact?
3. WHEN can this question be answered?
4. HOW does the answer to this question impact the architecture of the system?

# What is software architecture – My opinion

Oddly, this isn't really captured by most books

- (The original Shaw and Garlan book is probably closest)

1. The design of the entire system implementation (as envisioned)
  - Which may include hardware components, as well as networking/communication elements
  - Can be either an existing or proposed system
2. Includes partitions of the system
  - Some are physical (see above), some are purely software
  - If you have partitions, you must have connections between them
3. Abstract
  - Few, if any, details. This is intentional
4. Informal
  - Goes along with abstract
  - At odds with a lot of folks (mainly academics who haven't actually designed a system ...)
5. It's all about conveying big-picture information
  - And providing guidance and a conceptual framework for detailed design and implementation
  - Which implies that architecture is intimately tied to its documentation (but see next slide)

# Documentation, descriptions, etc.

Diagrams and views of the architecture are NOT the architecture

- Just like a UML diagram is NOT classes

The real architecture is in:

- The bits (if the system already exists)
- Our head (if it doesn't exist, and even if it does)
- (forward reference: programming as theory building)

But we often refer to the documentation as “the architecture”

- Wrong, but convenient...

# Fundamentals of Software Architecture

Three basic understandings

# Fundamental Understanding: 1

## Every Application has an Architecture

Not every architecture is good!

Not every architecture is intentional

If an architecture is not intentional, where did it come from?

What are the properties that make an architecture good or bad?

# Fundamental Understanding: 2

**Every application has at least one architect**

Why is it important that we understand this?

What are the implications of this understanding?

# Fundamental Understanding: 3

**Architecture is not a phase of software development**

Where does this incorrect notion come from?

First: the idea that design of software is an activity that strictly precedes coding

Second: the idea that high-level design (called architecture) is an activity that strictly precedes low-level design

# Architecture-Centric Software Development

Some people claim that software development should be architecture-centric.

I claim that software development IS by nature architecture-centric (if done rationally).

- In other words, software development is strongly influenced by architecture
- And in many cases, influences the architecture

If I'm right, what are the implications?

# Implications:

You can't design the architecture once and be done  
But if it affects everything, upfront thought (and design) is necessary

- A good architecture makes development (and maintenance) easier
- A bad architecture makes development and maintenance harder
- Maybe that's how we define a “good” or “bad” architecture! (see next slide)

If the architecture affects everything, it is important that everyone understand the architecture

- What does that say about the role of architecture?
- And the role of the architect?
- Naur: Programming as Theory Building

# Implications #2

Good or bad architecture:

- In some cases, it's that a particular architecture is inappropriate for the particular system.
- A different architecture would make implementation of the system easier.
- We will explore this later with respect to architecture patterns and quality attributes.

# Requirements

How are requirements important to the architecture?

How is the architecture important to requirements?

# Design

Where is the line between the architecture and the design?

The process of design is a continuum:

- Sometimes you are at a system level
- Sometimes you are at a high level
  - (high level is not always at a system level)
- Sometimes you are at a detailed level

Early on, it's mostly system and high level

- Later, it's mostly detailed

Answer: the line is wherever we think it makes sense to draw it.

# Coding, Testing, Deployment

How does architecture impact these?

How do these impact the architecture?

“Architect Also Implements”

# Views of the Architecture Process

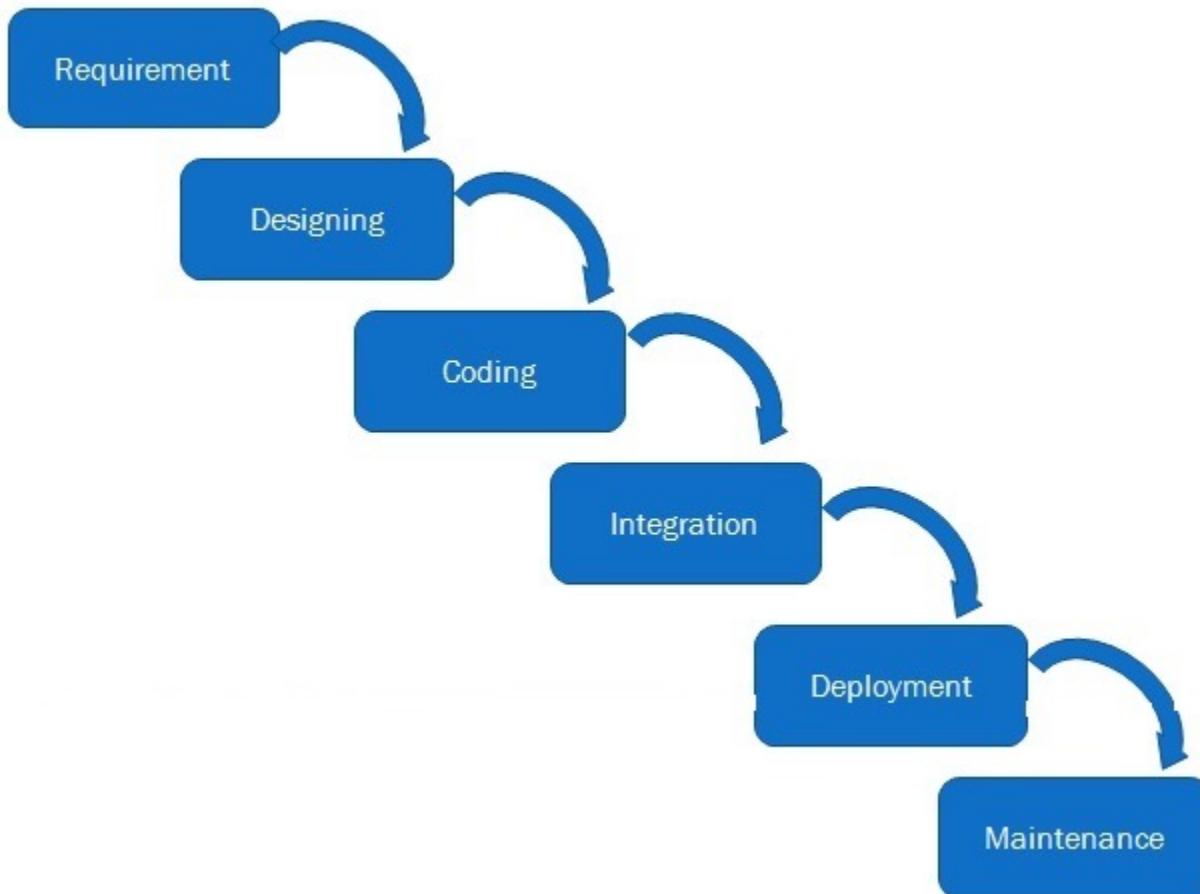
(a.k.a. the high level design process)

Two Extreme Views:

Traditional waterfall

XP: “Good design [architecture] emerges through refactoring” (Martin Fowler; also embraced by Ron Jefferies)

# Architecture and Waterfall



# Waterfall:

What's wrong with the waterfall view of design?

Designing and coding are iterative, highly integrated

If you think it of a model of time, it's wrong

What's RIGHT with the waterfall view of  
(architectural) design?

You need a general architecture beforehand in order  
to guide detailed design and coding.

It does show (part of) an information dependency  
model.

# XP and TDD

Big Design Up Front (BDUF) is BAD

Good Architecture emerges through refactoring

Focus on unit tests and refactor to create a good design

What's wrong with this approach?

It's a recipe for accidental architecture; it doesn't acknowledge the need for intentional architectural design

- User stories don't live in a vacuum; they live in context of the entire system
- And architecture is all about that context

It violates the Second Law of Thermodynamics (i.e., it's a fantasy)

What's right with this approach?

Basically nothing

# **Goldilocks: How Much Architecture?**

What's wrong with comprehensive architectural design up front?

What's wrong with letting the architecture emerge from the code?

What constitutes the “just right” level?

In practical terms,  
it's a question of  
when do we start  
coding

Here's one view:

When CAN you  
start coding?



# Goldilocks (again)

You can start coding (something) when you:

- Have a good (abstract) picture of that part of the system.
  - And ...
- Have confidence that it (the picture) won't change a lot.

Or...

- You want to learn something

In other words:

- You can start some coding pretty early

BUT

- By definition, design precedes coding

Bottom line:

- The Waterfall folks aren't completely wrong, but aren't completely right.
- The XP folks aren't completely wrong, but aren't completely right.

# Pitfalls...



# Fallingwater (USA)

Designed by Frank Lloyd Wright, 1935

Unique, beautiful design

But plagued by problems:

- Structural stresses
- Building materials
- Leaks
- Mold

What went wrong?



# Lessons from Fallingwater

The architecture must consider practical issues:

- Building: construction, laws of physics, etc.
- Software: how will the packages, libraries, etc., play together, performance, deployment, etc.

You have to consider the lifetime of the product:

- Building: will stresses cause problems later?
- Software: how easy is it to maintain? Will it evolve gracefully?

Architects do not live in ivory towers

But imagine how bad it would have been without a guiding architecture!

# Quality Attributes

Fallingwater: Things like the ability to withstand stresses over time

We have lots in software

- We will talk a lot about them later

Quality attributes are system-wide properties

- Which leads to a problem...

# Software Quality Attributes

When do you know for sure that:

- Your software has sufficient performance?
- That it scales up easily?
- That it is secure against all types of security breaches?
- That it recovers gracefully from all types of faults?
- That it runs nonstop for months?

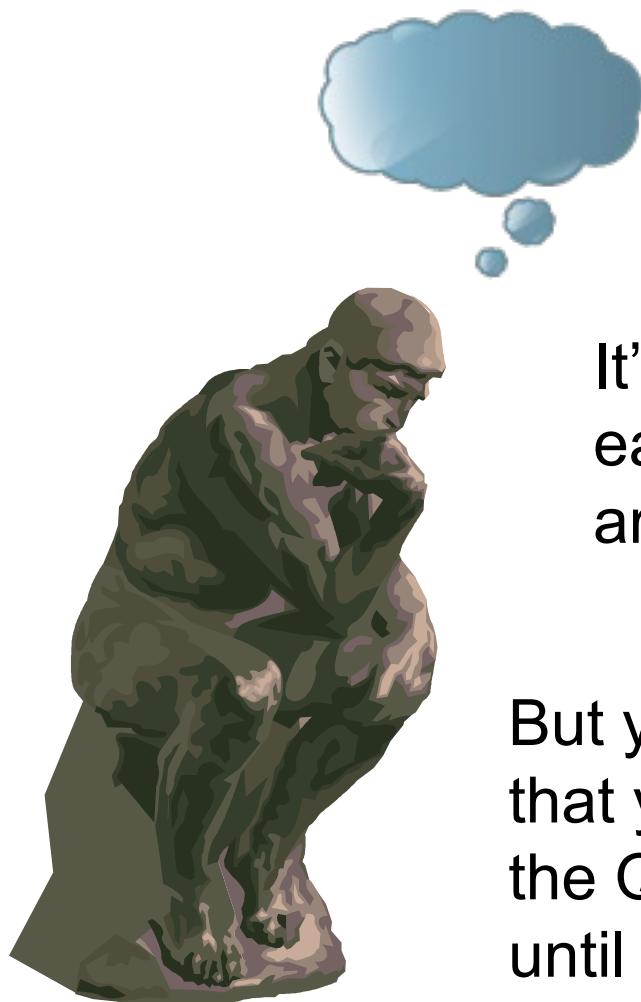
In general, at system test time. Why?

- Because you need to test the entire system
- In these aspects, the code is as strong as its weakest link

# The Architect's Dilemma

Quality Attributes  
are strongly  
influenced by the  
Architecture

And the  
architecture is  
designed EARLY  
in the project



It's too late to  
easily change the  
architecture

But you don't know  
that you achieved  
the Quality Attributes  
until the system is  
complete!

# Finding out Too Late ...

## The Vasa Warship 1628

- Sank on maiden voyage, less than 2km from shore
- Architecture did not support the top-heavy ship
- System test: 30 men ran from side to side on deck – Failed Stability Test!



# Side Lesson from the Vasa

What does it say about changing requirements after the architecture is complete?

- (careful here!)

# Conway's Law

The structure of the system and the structure of the committee inventing the system are isomorphic.

What does this mean?

# Conway's Law

How does this apply to software projects?

Which comes first, the team or the architecture?

# Design Exercise!

Divide into teams. Each team gets a paradigm:

Your paradigm:

Create a design for the following:

- The program inputs a positive integer, and outputs the equivalent Roman Numeral.

10-15 minutes: present a description (pictures, flowcharts, pseudo-code, or whatever you want)

$1 = I$

$4 = IV$

$5 = V$

$9 = IX$

$10 = X$

$11 = XI$

$14 = XIV$

$15 = XV$

$50 = L$

$100 = C$

# Table driven (not strict): trivial

```
romn = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1]
roms = ['M', 'CM', 'D', 'CD', 'C', 'XC', 'L', 'XL', 'X', 'IX', 'V', 'IV', 'I']
```

```
def rom(n):
    numeral = ""
    index = 0
    while (n > 0):
        while (n >= romn[index]):
            numeral = numeral + roms[index]
            n = n - romn[index]
        index = index + 1
    return numeral
```

```
for i in range(2000, 2026):
    print(rom(i))
```

# What do we learn?

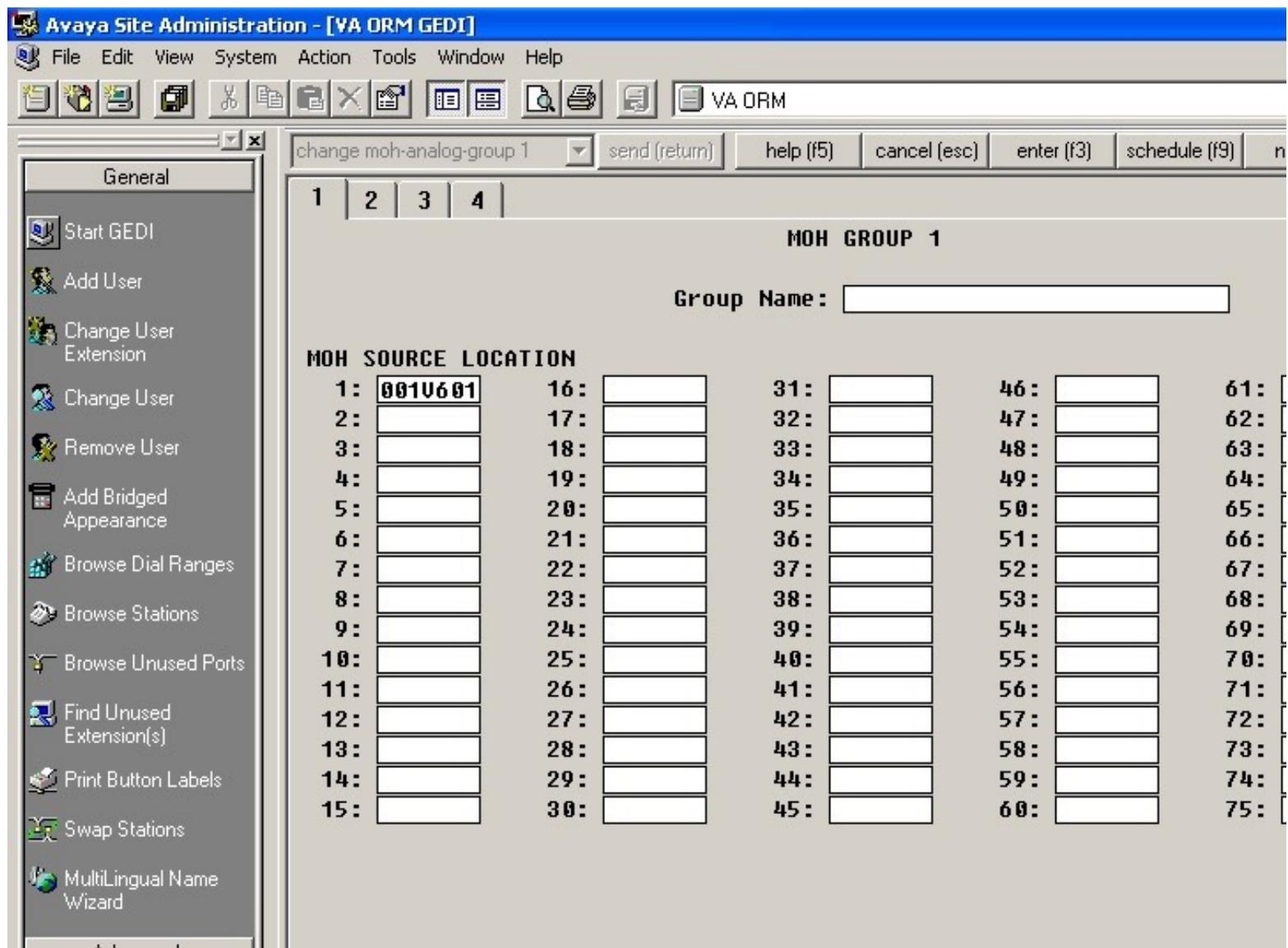
For a given problem, some approaches work better than others

- The programming paradigms kind of forced some partitions (especially the OO paradigm)
- ... which may or may not match the natural partitioning of the solution

**A major part of architectural design is deciding on the primary (high level) partitions**

- Many are logical (classes, functions, data entities, etc.)
- Some can be physical (e.g., client/server)

# Table driven: switching system example



# OO: the Paradigm du jour

Dominant approach these days

What are the advantages?

But what are the disadvantages?

# Basic Concepts in Software Architecture

## Terminology

- Architecture
- Reference Architecture
- Descriptive vs. Prescriptive Architecture
- Component
- Connector
- Architectural Style
- Architectural Pattern

## Models

## Processes

## Stakeholders

# Architecture Definitions (from books)

“The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.” Perry & Wolf (probably the best definition)

“A software system’s architecture is the set of principal design decisions made about the system.”  
(from the textbook)

“Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.” – Eoin Woods

# Architecture as a set of Design Decisions?

One popular notion

It's general – more than the structure of the system

Design decisions include those related to:

- System structure
- Functional behavior
- Interaction
- Non-functional properties
- Implementation

Just about everything is a decision

I don't particularly like this notion of architecture

- It doesn't give me a good system-wide view
- But it is still a useful concept

# Reference Architecture

A single software architecture for a family of related software systems

“A reference architecture is the set of **principal** design decisions that are **simultaneously** applicable to **multiple related systems**, typically within an **application domain**, with explicitly defined **points of variation**.”

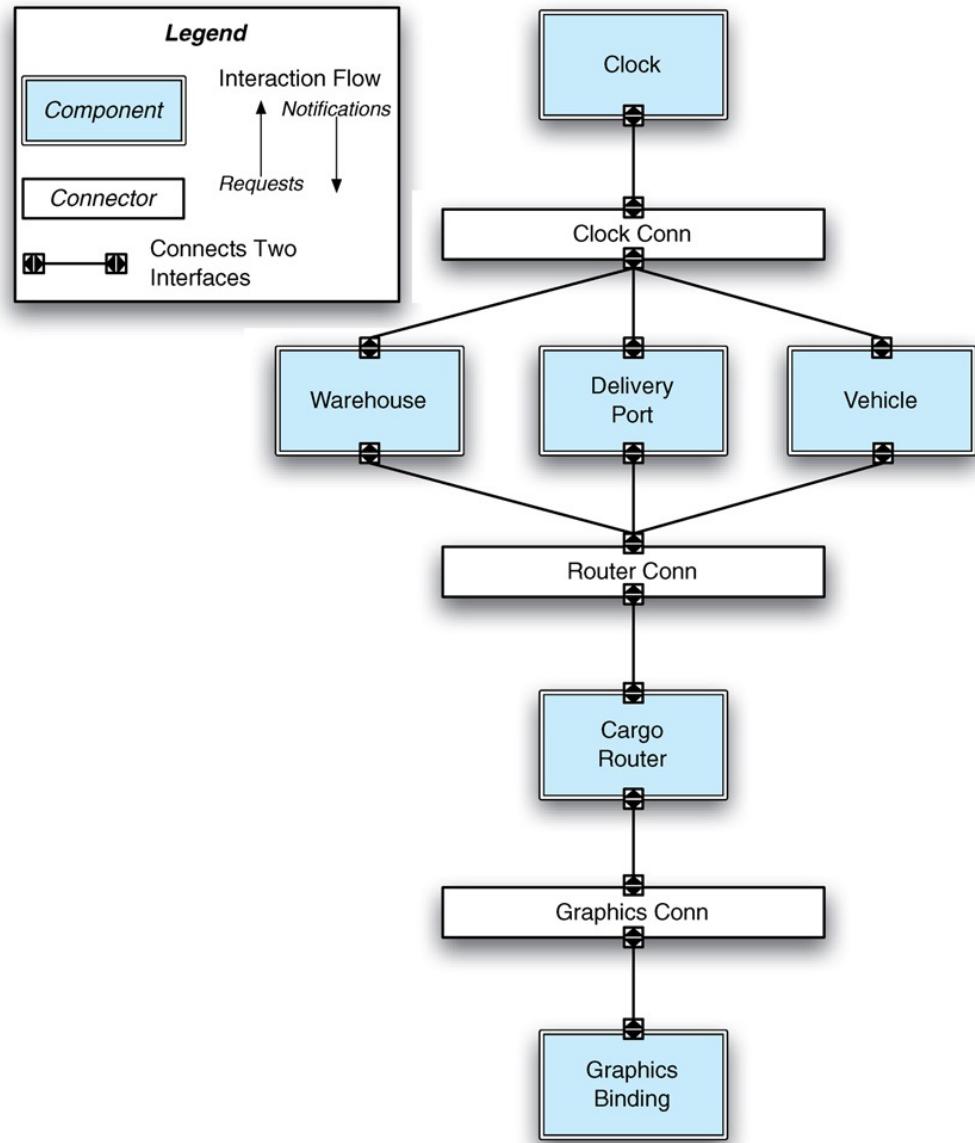
# **Descriptive vs. Prescriptive Architecture**

## **Descriptive:**

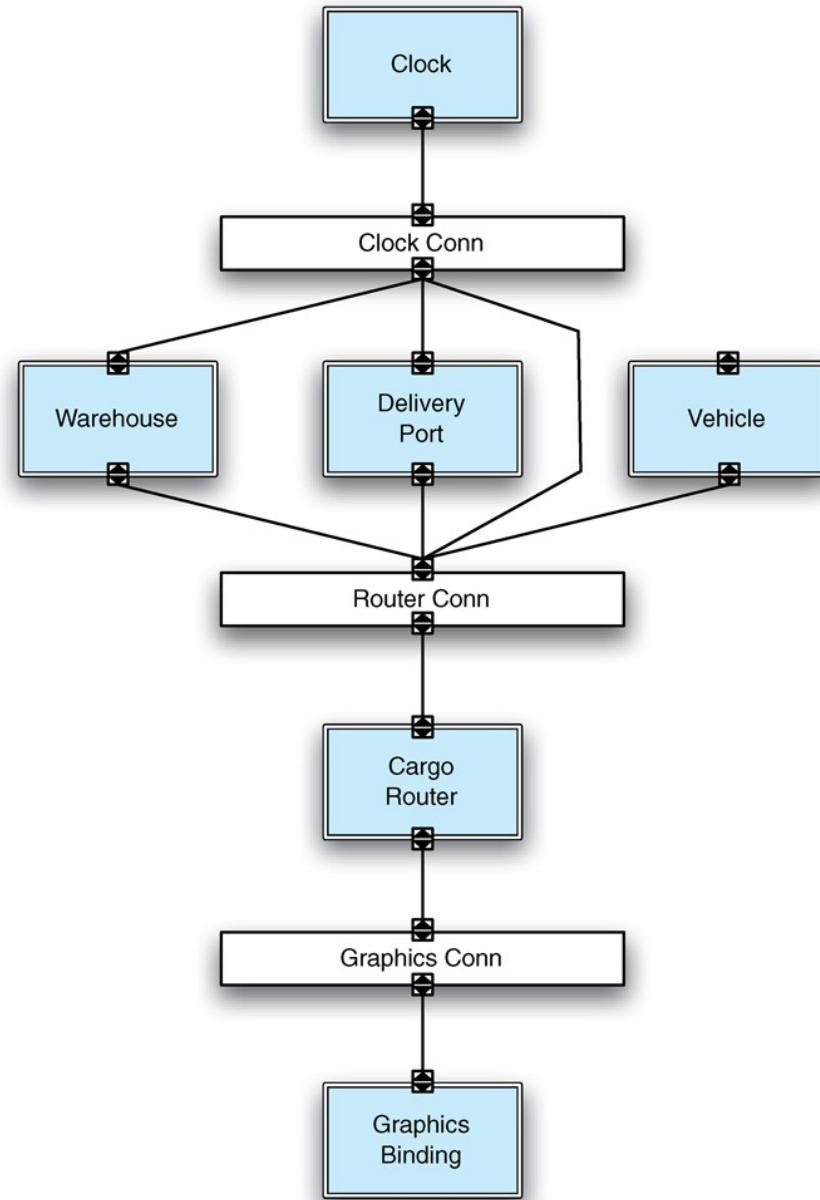
- Describes what is there
- Purposes:
  - Convey understanding of how a system is organized
  - Convey rationale for how the system is organized
  - Convey the theory of the program (includes the previous two)
  - Aid in maintenance and enhancement of the system
- Prescriptive
  - Describes the structure of the system to be created
  - Establish the theory of the program
  - Specifies some technical/development issues

# Prescriptive: as imagined

A cargo routing application



# Descriptive: what was implemented



# Architectural Degradation

## Architectural Drift:

- The introduction of principal design decisions into a systems descriptive architecture that are:
  - Not included in, encompassed by, or implied by its prescriptive architecture
  - Don't violate its design decisions

## Architectural Erosion:

- Introduction of architectural design decisions into a systems descriptive architecture that:
  - Violate its architecture

## My interpretation:

- Architectural drift moves the architecture
- Architectural erosion destroys it

Caveat: most literature doesn't distinguish, or calls everything "architectural drift".

Caveat #2: How do you distinguish between Drift and Erosion?

# Component

(book): A software component is an architectural entity that

- Encapsulates a subset of the system's functionality and/or data
- Restricts access to that subset via an explicitly defined interface
- Has explicitly defined dependencies on its required execution context.

Another way of looking at it:

- A software component is a locus of computation and state in a system

Key aspect:

- You can “see” a component from the outside (and kind of ignore its innards)

Note: “component” is hard to pin down. The book’s definition is as good as I’ve seen.

# Connector

(book): A software connector is an architectural element tasked with effecting and regulating interactions among components

(Note: “effecting” means “to make happen”.)

Multiple types of connectors

Their importance is largely underappreciated

# Architectural Styles, Patterns

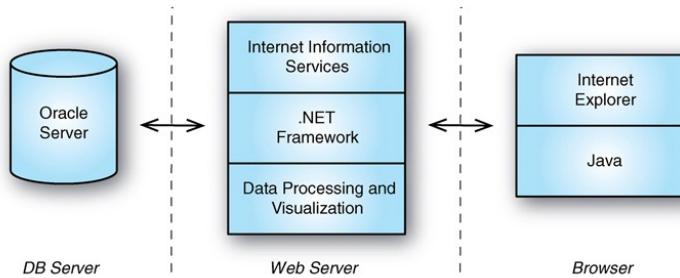
Some confusion about these terms: considerable overlap

Both are names for general architectural shapes

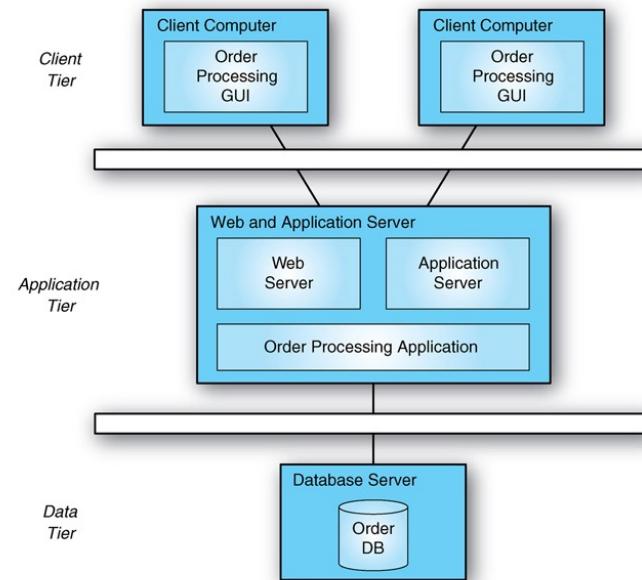
My opinion: architectural styles are broad architectural approaches, while patterns are often more context-specific.

# Examples of 3-Tier Architecture pattern

Generic view:



Specific application: an order processing system



# Architectural Model

(book): “An architectural model is an artifact that captures some or all of the design decisions that comprise a system’s architecture.

Architectural modeling is the reification and documentation of those design decisions.”

(mine): An architectural model is a representation of the architecture. It’s usually expressed as a combination of text and pictures (although the text and pictures are NOT the model, but a description of it.) It helps us reason about the system (not just about the system’s architecture.)

(See Wikipedia)

# References

Naur: Programming as Theory Building

Parnas: The Modular Structure of Complex Systems

- (Really what architecture is all about)

# Architectural Recovery

(book) “Architectural recovery is the process of determining a software system’s architecture from its implementation artifacts.”

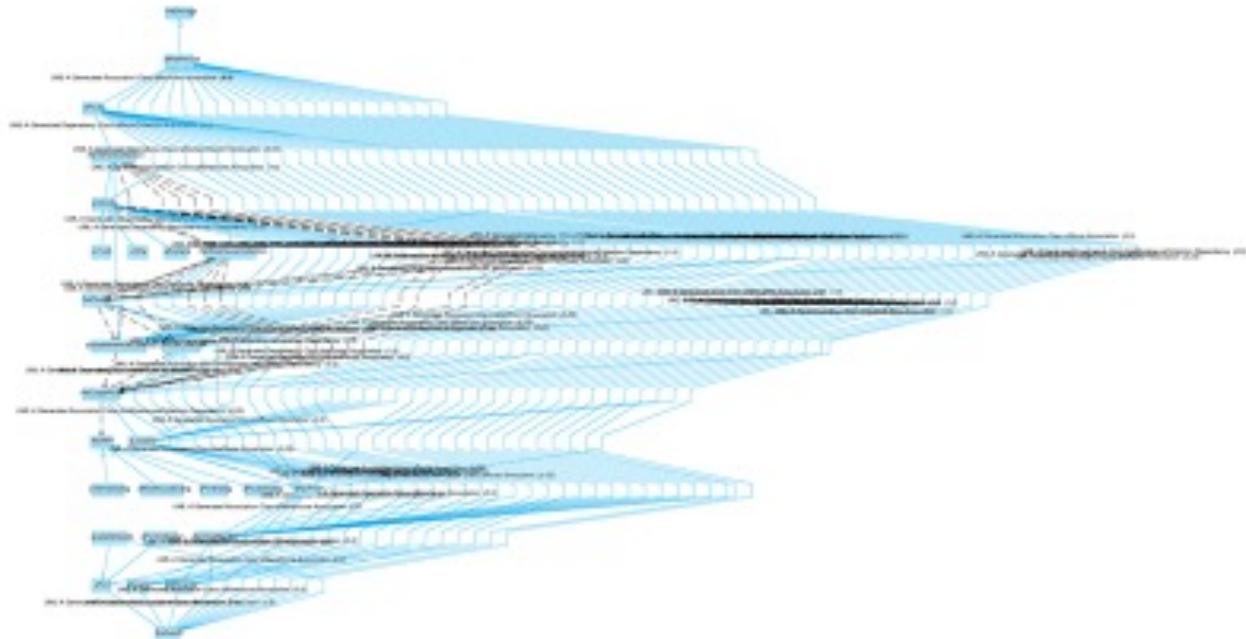
Artifacts include code and documents

By definition, it extracts a system’s descriptive architecture

- Note: It’s really hard to extract the theory of the program from the code alone

Architectural recovery tools may help (or not):

# Find the Architecture



fig\_03\_06

# Architectural Recovery

How would you go about it?

What information will you try to find?

Where to look?

(Assume a lack of architecture or design documentation)

# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**

# Bowling: what did you learn?

How well did the procedural paradigm fit?

- Why?
- What types of problems fit well with procedural programs?

How well did the OO paradigm fit?

- Why?
- What types of problems fit with OO designs?

How well did the Pipes and Filters fit?

- Why?
- What types of problems fit well with P&F?

Change happens!

- What if the program supports multiple players taking turns?
- What if you add error checking (and report at the end?)

# Designing Architectures

The big question!

1. The Design Process
2. Architectural conception
3. Styles and Patterns
4. Architectural conception in a vacuum
5. Putting it all together

# How do you design software?

Let's observe it happening in the wild! 😊

Groups of Two teams:

- One team: design a software system (See Spec)
  - Work together around a board
  - Write your ideas/architecture
- The other: watch everything they do:
  - HOW do they design software?
  - What activities, in what order?
  - Write down what you see

What did you observe?

# What you might have observed

Trying to break the system into pieces

- Drawing boxes for the pieces
- Getting it wrong and erasing some boxes

Discussing quality attributes

- Like reliability or performance

Discussing how the pieces communicate with each other

- Should happen, but often doesn't!

Discussing external software to use

Lots of questions that aren't answered

- Should write them down, usually don't

Discussed some things in detail

- Sometimes too much detail (right now)

Often, one person becomes dominant in a group

- Dominates the conversation
- Might also hold the marker most of the time

# Essential Activities in System Design

Learn the functional requirements

- (of course!)

Understand the non-functional requirements

- (Thought question: why is this important at the architectural level?)

**Architectural Conception**

- First ideas of the architecture

Create the physical layout of the system

- If hardware is involved (beyond one computer)

High-level partitioning

- Design chunks of functionality (components)
- Understand how the components interact (what are the connectors?)

Consider technology alternatives

Note: the above list is not ordered, and it's iterative

# Architectural Conception

The first, very general, concept of what the system design (the architecture) will look like

## Mind Tools:

- Partitioning, separation of concerns
- Abstraction
- Simple machines
- Prior knowledge (Refined experience)

## Physical tools

- Whiteboard
- Cards (similar to CRC cards)
- Your colleagues 😊
- Some visualization tools

## Tools that don't work:

- Software tools, such as CASE (Computer-Aided Software Engineering)
- Code generators (wrong level: too detailed)

# Weighing Alternatives

Many Architectural Decisions consist of selecting one choice among several alternatives

Particularly true during Conception

- But happens throughout architecture (and development)

Technology Alternatives

- Please give examples

Non-Technology Alternatives

- Please give examples (and why are they architectural?)

# Partitioning

The basic action of architectural design

Why?

- Because we can't hold the whole design in our head
- Separation of concerns, reduce coupling
- Convenience for development, multiple people can work on it
- Can isolate/reduce errors
- Might find some parts already done
- Might be able to reuse parts elsewhere
- Ease in maintenance and extension: easier to understand

# Separation of Concerns

You can partition a system millions of ways!

So the big questions are:

- What is the best partitioning?
- What makes one partitioning better than another?
- What criteria should you use to partition?

References:

- David Parnas, “On the Criteria to be Used in Decomposing Systems into Modules” (1972!)
- David Parnas, Paul Clements, David Weiss: “The Modular Structure of Complex Systems” (1985)



# Characteristics of Good Modules

Easier to understand:

- The modules make intuitive sense
- We have a high-level picture; not distracted by the weeds

Modifications to the system typically involve few modules

- Horror story: telephone system

The interfaces between the modules change relatively infrequently

Low coupling

High cohesion (Everything in a partition should be about the same thing: Single Responsibility Principle)

# Modules and Hardware

Many modern systems involve multiple hardware devices

- Hardware units are natural modules
- Or distributed across networks

Some partitions are obvious:

- An app on your phone typically has a phone piece and a server piece
- The obvious hardware partitioning is a good starting point

Some may be less obvious, or even our choice

- We may weigh factors such as cost, performance, capacity, etc.
- And what makes sense to us!

# Modules and Data

Data can guide us to partition into modules

Common:

- For persistent data, a module that manages the data (i.e., a database)

Also consider putting operations on a common set of data together into a module

- What examples can you think of?

# User Interactions and Modules

User interactions can guide partitioning

Can have modules that deal with the user interaction

Can have different modules for different types of users

# Responsibilities and Modules

When partitioning, consider that each module:

- Ideally has ONE major responsibility
- (so you can identify the major responsibilities and make a component for each)
- Of course, the major responsibility may have sub-responsibilities

Can also communicate with other modules

Can model it by writing on cards

- Similar to CRC (Class, Responsibility, Collaborator) cards
- But at a higher level of granularity

Much of the software architecting activity is related to partitions:

- What are the modules?
  - What are the responsibilities of each module?
  - What are the connections between module?
- 
- Does each scenario of usage actually work in the proposed partitioning?

# Scenarios

Walk through usage scenarios to validate the partitioning

You can do it once you have a first, rough cut at partitioning

# Abstraction

The structured removal (or hiding) of information

Abstraction can encompass sets of closely related things that can be considered generically (think Base Classes)

Or can be a general idea about something, without considering the details

# Side note: Diving into details

As you design an architecture, you work mainly at a very high level

Occasionally, you go into more detailed design

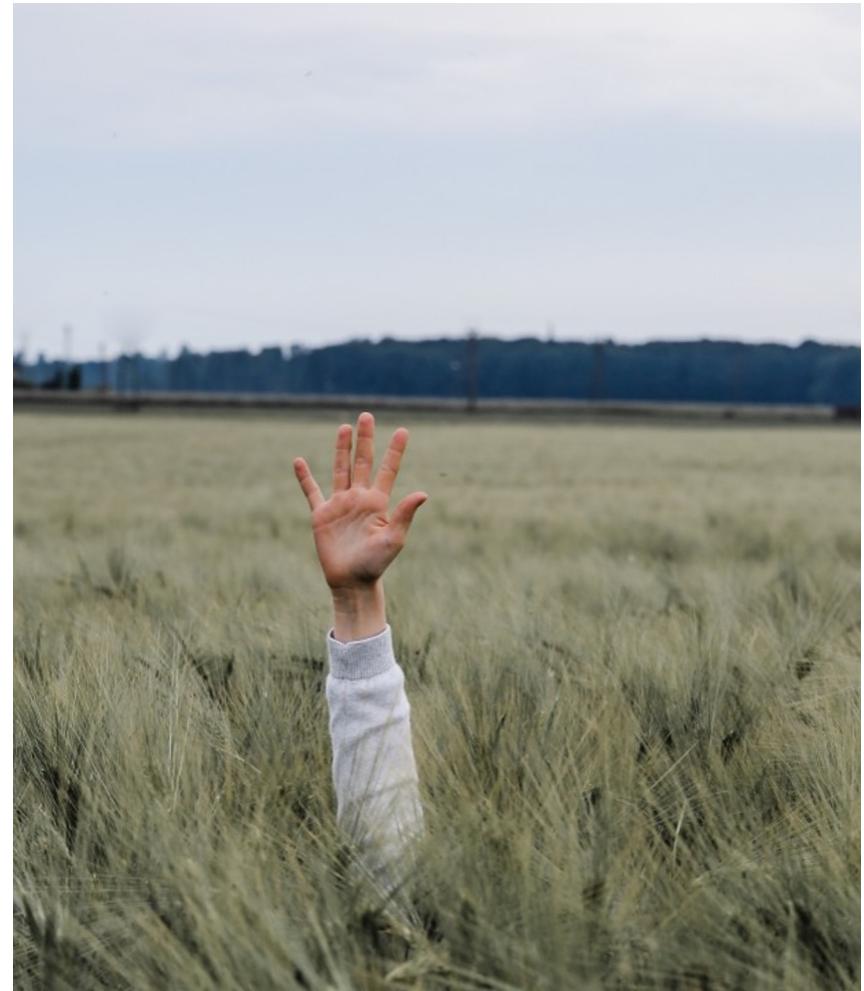
- It might answer some question. For example, you might wonder how a particular design might meet a non-functional requirement. You might have to dive down into more detail

Then pop back up to high level

Don't get stuck in the weeds (the details)

How deep is too deep?

- For example, the data and methods of a given class are “the weeds”.



# A Continuum of Detail?

What's the difference between high-level design and detailed design?

Is it just a matter of scale and levels of detail?

Yes and no

Some design is focused on partitioning

- This is mainly large scale and high level
- But it can be small scale and fairly detailed

But some design is algorithm creation

- Pretty much all low-level and detailed

Architecture is *mainly* concerned with the partitioning

# Domain-Specific Software Architecture

Just what it sounds: an architecture for multiple similar applications in a particular domain

Pretty close to product families

- Reference architecture of a product family is a specialization or a subset of a Domain-Specific Software Architecture

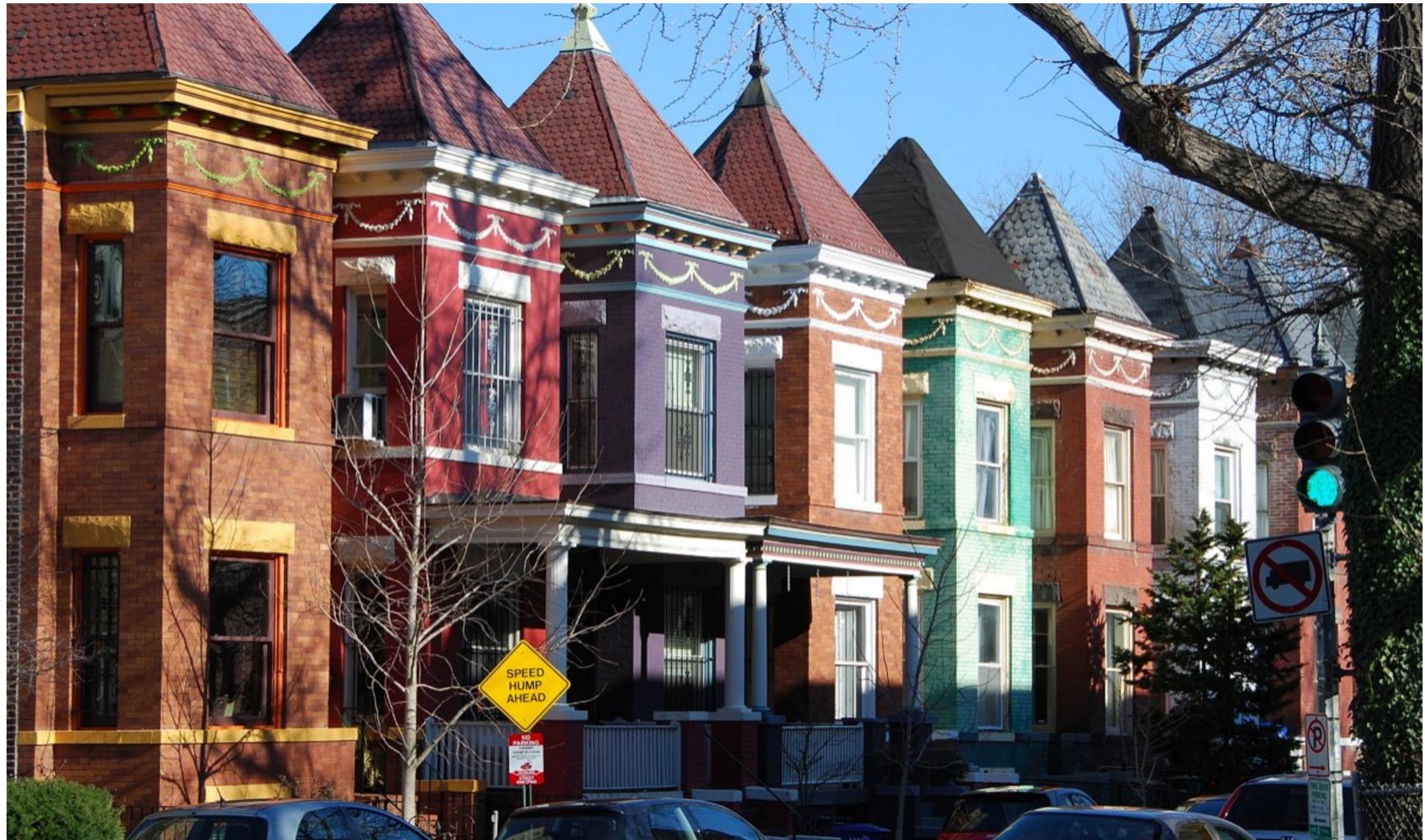
A combination of:

- A reference architecture for an application domain
- A library of software components for that architecture containing reusable chunks of domain expertise
- A method of choosing and configuring components to work within an instance of the reference architecture

Building analogy:

- Row houses: not identical to each other, but similar
  - Easy to build
  - Easy to customize

# A Row of Houses



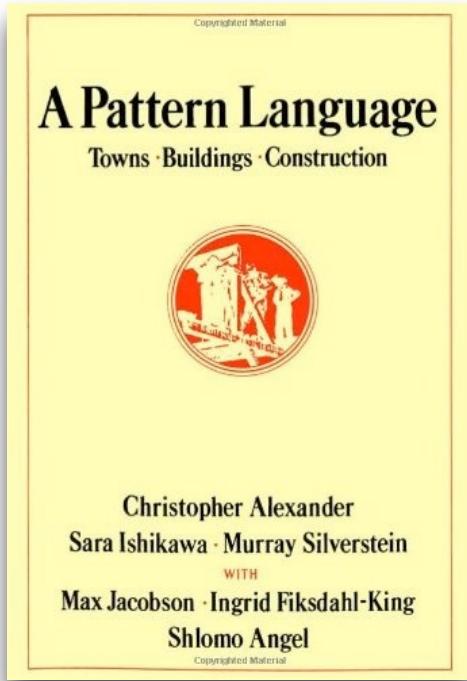
# **Software Architecture**

FEUP-M.EIC-ASSO-2024

**Ademar Aguiar, Neil Harrison**

# **Patterns of Software**

# The Origins of Patterns, 1977



- Christopher Alexander in this book presents a pattern language, an ordered collection of 253 patterns.
- The goal was to enable non-experts to architect and design their own houses and communities with quality.
- Patterns describe current practice and provide vision for the future.
- Patterns integrate knowledge from diverse sources and link theory and practice.
- Patterns have a strong “bottom up” orientation.

<http://c2.com/cgi/wiki?ChristopherAlexander>

# What is a pattern?

The elements of this language are entities called patterns. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

from *A Pattern Language*, Alexander et al, 1977

# **253 patterns**

## **from global to particular problems**

- 1. Independent Regions
- 2. The Distribution of Towns
- 16. Web of Public Transportation
- 18. Network of Learning
- 20. Mini-buses
- 43. University as a Marketplace
- 83. Master and Apprentices
- 134. Zen View
- 253. Things from your life

**title**

**context**

## 20 MINI-BUSES\*

. . . this pattern helps complete the LOCAL TRANSPORT AREAS (11) and the WEB OF PUBLIC TRANSPORTATION (16). The local transport areas rely heavily on foot traffic, and on bikes and carts and horses. The web of public transportation relies on trains and planes and buses. Both of these patterns need a more flexible kind of public transportation to support them.

**problem**



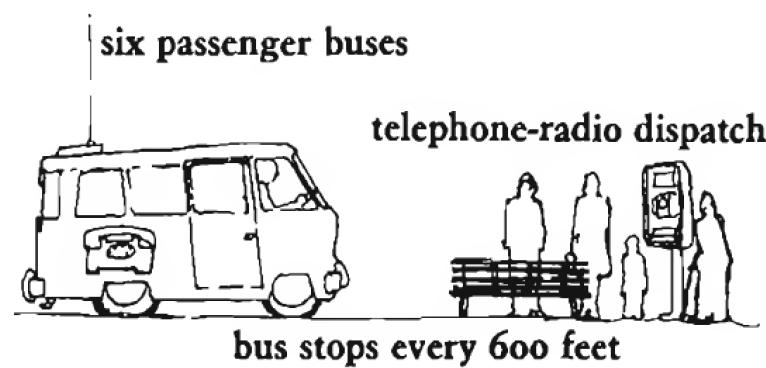
Public transportation must be able to take people from any point to any other point within the metropolitan area.

**discussion**

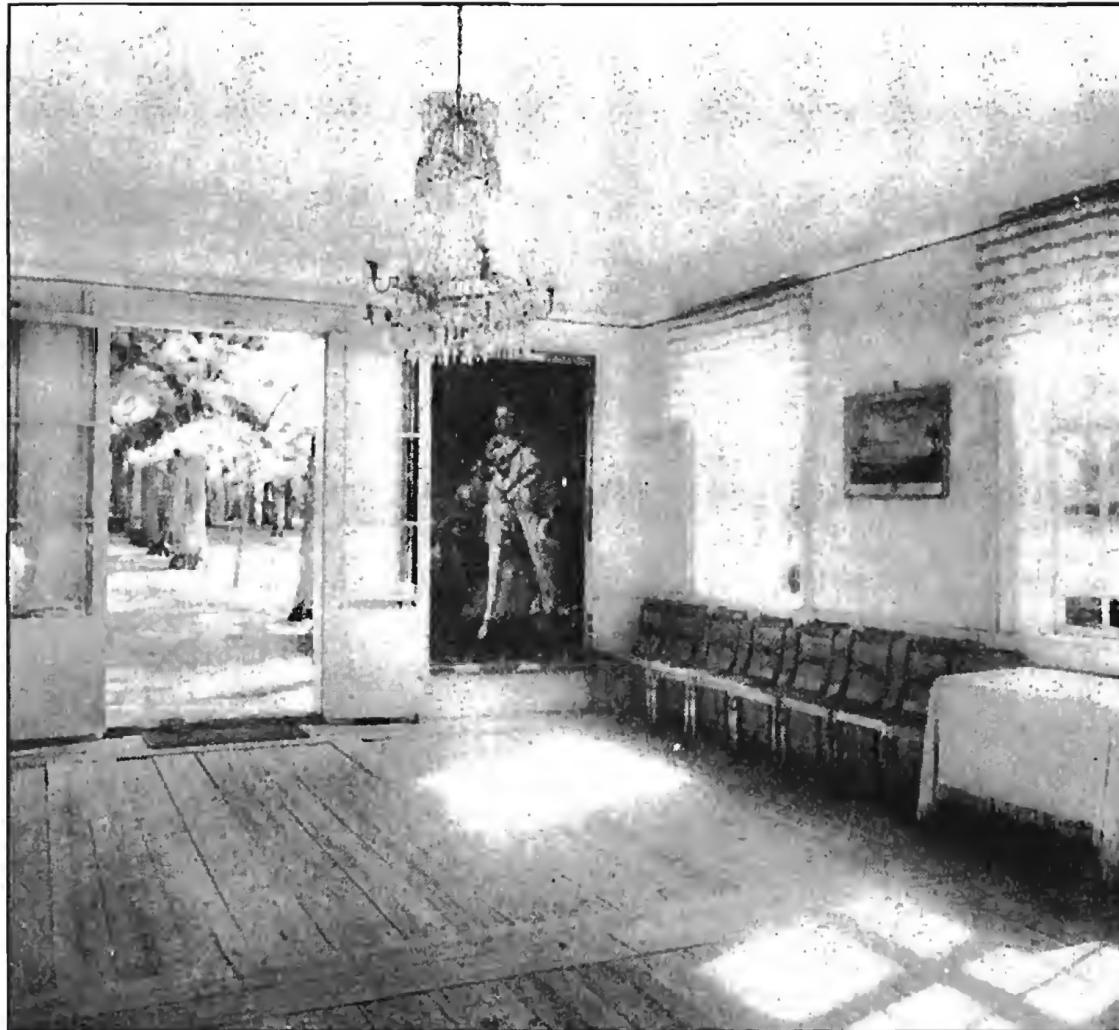
Buses and trains, which run along lines, are too far from most origins and destinations to be useful. Taxis, which can go from point to point, are too expensive.

## solution

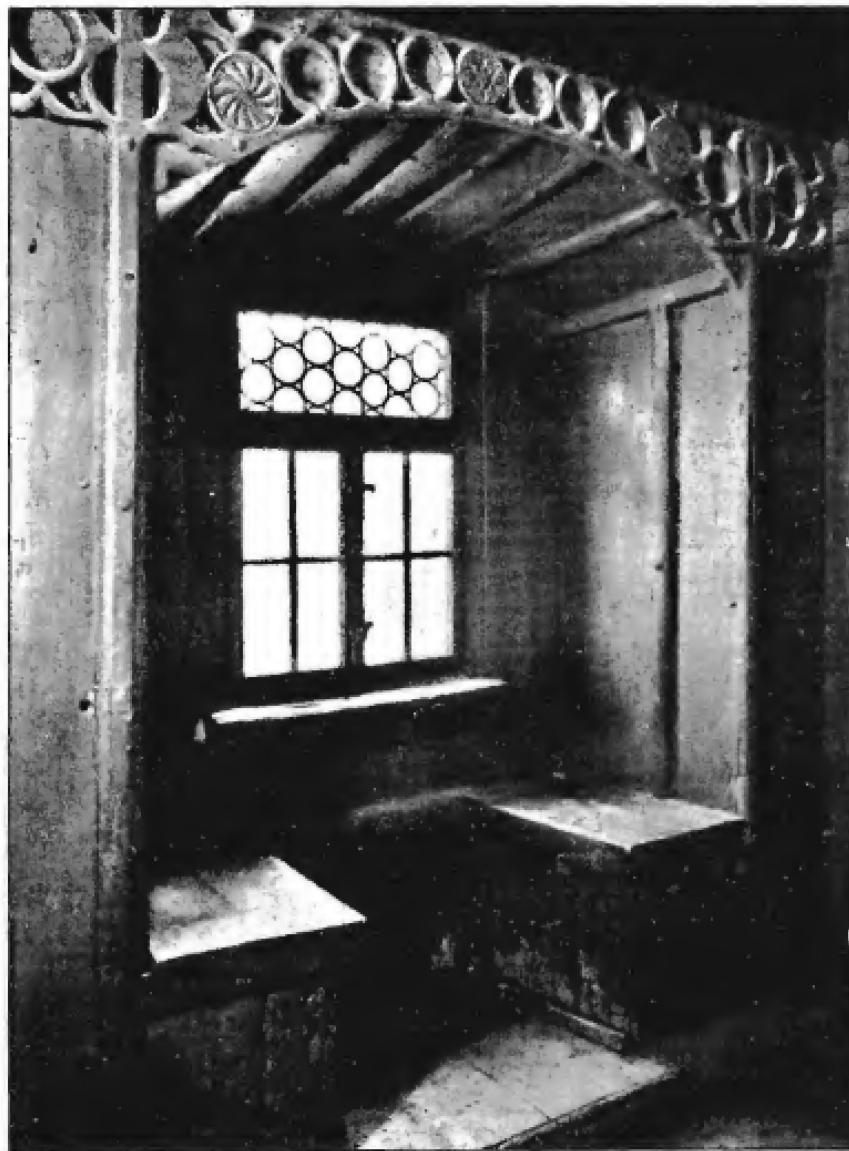
Establish a system of small taxi-like buses, carrying up to six people each, radio-controlled, on call by telephone, able to provide point-to-point service according to the passengers' needs, and supplemented by a computer system which guarantees minimum detours, and minimum waiting times. Make bus stops for the mini-buses every 600 feet in each direction, and equip these bus stops with a phone for dialing a bus.



I 59 LIGHT ON TWO SIDES  
OF EVERY ROOM\*\*



I 80 WINDOW PLACE\*\*



# 251 DIFFERENT CHAIRS



253 THINGS FROM  
YOUR LIFE\*



# QWAN

Quality Without A Name

## Fifteen Properties

The degree of life which appears in a thing depends upon the life of its component centers and their density. Following are fifteen structural features which he has identified as appearing again and again in things which have life.

Together, these fifteen properties identify the character of living systems. They are the principal ways in which centers can be strengthened by other centers. They are not independent, but rather rely on and reinforce each other. Things which are more whole, which exhibit more life, will have these fifteen properties to a strong degree. Conversely, the things in this world which are most lifeless will have these properties to the least degree.

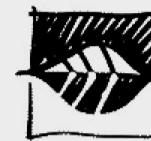
1. Level of Scale



5. Positive Space



9. Contrast



13. The Void



2. Strong Centers



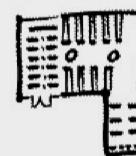
6. Good Shape



10. Graded Variation



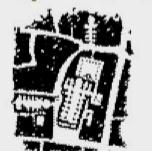
14. Inner Calm



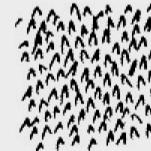
3. Boundaries



7. Local Symmetries



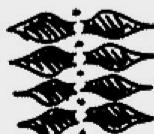
11. Roughness



15. Not-Separateness



4. Alternating Repetition



8. Deep Interlock



12. Echoes



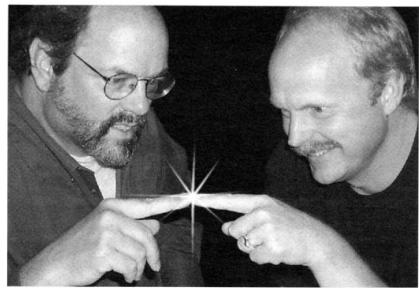
# Quality attributes in Software Architecture

Supportability	Modifiability	Scalability	Reusability
Maintainability	Testability	Availability	Security
Interoperability	Usability	Reliability	Performance



<https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>

# Software Patterns, 1987..1994..



A new software patterns discipline started by a group later called “The Hillside Group”, 1993.



Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (aka Gang of Four), published the first book on “Design Patterns”, the “bible”, 1994.



- Kent Beck & Ward Cunningham, “Using Pattern Languages for Object-Oriented Program”, OOPSLA '87, 1987.

# <http://wiki.c2.com/?HistoryOfPatterns>

The screenshot shows a web browser window with a title bar "People Projects And Patterns". The main content area features a decorative graphic on the left and the following text:

**People Projects And Patterns**

"Who -- People, What -- Projects, and How -- Patterns"

---

**People.**

People who are important to the practice of Software Development, that is... Not every famous philosopher belongs here; perhaps not even every software developer who has written a book. But if someone has said something important on the topic of Software Development or Patterns, by all means, tell us what important things they said.

On people pages - we describe individuals like Christopher Alexander or Kent Beck. People don't always write their own pages. There are too many noteworthy people to expect that to happen. Likewise, don't take what's there too seriously. If you find something you know to be wrong or inappropriate, take the time to edit it. Be kind and use understanding as some folks are new to this.

- [PeopleIndex](#)

**Projects.**

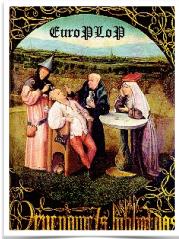
We believe patterns spread from person to person as they work together on projects. Just what are those projects? Look for project pages like [MacApp](#) or [HotDraw](#) or [SmalltalkSummer](#). Don't look for secrets, and don't write any yourself.

- [ProjectIndex](#)

**Patterns.**

Now these are the real gems. We're looking for that common knowledge that's so uncommon. For example, [CommandObject](#) makes undo and redo easy while [WindowPerTask](#) addresses updating issues in early [ModelViewController](#) (MVC). [ModelRendererView](#) describes a variation on the theme. These pages won't necessarily contain the usual parts of a [WrittenPattern](#). We're just labeling ideas so we can study how they flow.

# PLoP Conferences



Friday, Jul 07, 2017

www.hillside.net/conferences

Pattern Languages of Programs (PLoP) Conferences

[Login](#) [Text Size](#)

 THE HILLSIDE GROUP

Hilltop Books Contact Conferences Patterns Vision Wiki

**MAIN MENU**

- ▶ Hilltop
- ▶ Books
- ▶ Contact
- ▶ **Conferences**
  - GuruPLoP
  - PLoP
  - Chili PLoP
  - EuroPLoP
  - Meta PLoP
  - Asian PLoP
  - ParaPLoP
  - Scrum PLoP
  - Sugarloaf PLoP
  - Viking PLoP
  - Other Conferences
  - Shepherding
  - Useful Documents
  - PLoP Conference Proceedings
  - PLoP Paper Template
- ▶ Patterns
- ▶ Vision

You are here: Home Conferences

## PATTERN LANGUAGES OF PROGRAMS (PLoP) CONFERENCES

 We have compiled our collected PLoP experiences into a series named [How to Run PLoP](#)

The Hillside Group Sponsors many different conferences such as: **PLoP**, **EuroPLoP**, **AsianPLoP**, **ScrumPLoP**, **VikingPLoP**, **SugarLoafPLoP**, **UP**, and **ChiliPLoP**. These conferences focus on writing groups to better improve patterns through group exposure. Each conference offers advanced topics for the more adept pattern writers. Participants have the opportunity to refine and extend their patterns with help from knowledgeable and sympathetic patterns enthusiasts.

[Conference Proceedings](#)



**PLoP**



Pattern Languages of Programs (PLoP™) conference is a premier event for pattern authors and pattern enthusiasts to gather, discuss and learn more about patterns and software development... [Learn More](#)

**CHILIPLoP**



The 13th Annual ChiliPLoP features "hot topics" for experienced folks... [Learn More](#)

**EURO PLoP**

**SUGARLOAF PLoP**



SugarLoafPLoP brings together researchers and practitioners whose interests span a remarkably broad range of topics, who share an interest in exploring the power of the pattern form... [Learn More](#)

# Patterns do and do not...

## Patterns do...

- provide common vocabulary
- provide “shorthand” for effectively communicating complex principles
- help document software architecture
- capture essential parts of a design in compact form
- show more than one solution
- describe software abstractions

## Patterns do not...

- provide an exact solution
- solve all design problems
- only apply for object-oriented design

# Patterns can be...

- non-generative (e.g. Gamma patterns)
  - observed in a system
  - descriptive and passive
- generative (e.g. Alexander patterns)
  - generate systems or parts of systems
  - perspective and active

# **Thank you!**

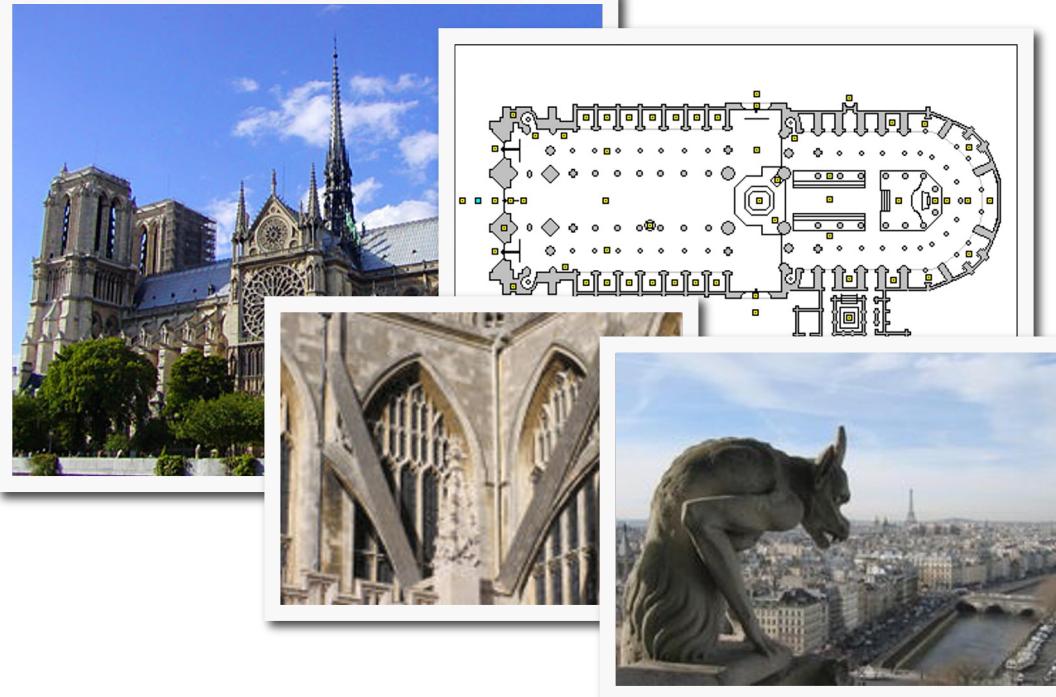
# **Software Architecture**

FEUP-M.EIC-ASSO-2023-24

**Ademar Aguiar, Neil Harrison**

# Architectural styles

## Early Gothic Architecture, Notre Dame, Paris



### Gothic characteristics

- Ogival archs, great expanses of glass, ribbed vaults, clustered columns, sharply pointed spires, flying buttresses and inventive sculptural detail such as gargoyles.  
[[http://en.wikipedia.org/wiki/Gothic\\_architecture](http://en.wikipedia.org/wiki/Gothic_architecture)]

# Styles and patterns

Identifying styles and patterns can help codify and share knowledge/expertise

## Architectural Styles

- [Shaw and Garlan, Software Architecture, Prentice Hall 96]

## Design Patterns

- [Gamma et. al, Design Patterns, Addison Wesley 95]
- [Buschmann et. al, Pattern-oriented Software Architecture: A System of Patterns, John Wiley & Sons 96]

## Code Patterns

- [Coplien, Advanced C++ Programming Styles and Idioms, Addison-Wesley 91]

# Architectural styles

An architectural style consists of:

- **a set of component types** (e.g., process, procedure) that perform some function at runtime
- **a topological layout of the components** showing their runtime relationships
- **a set of semantic constraints**
- **a set of connectors** (e.g., data streams, sockets) that mediate communication among components

Styles provide guidance for architectural design based on the problem domain and the context of use

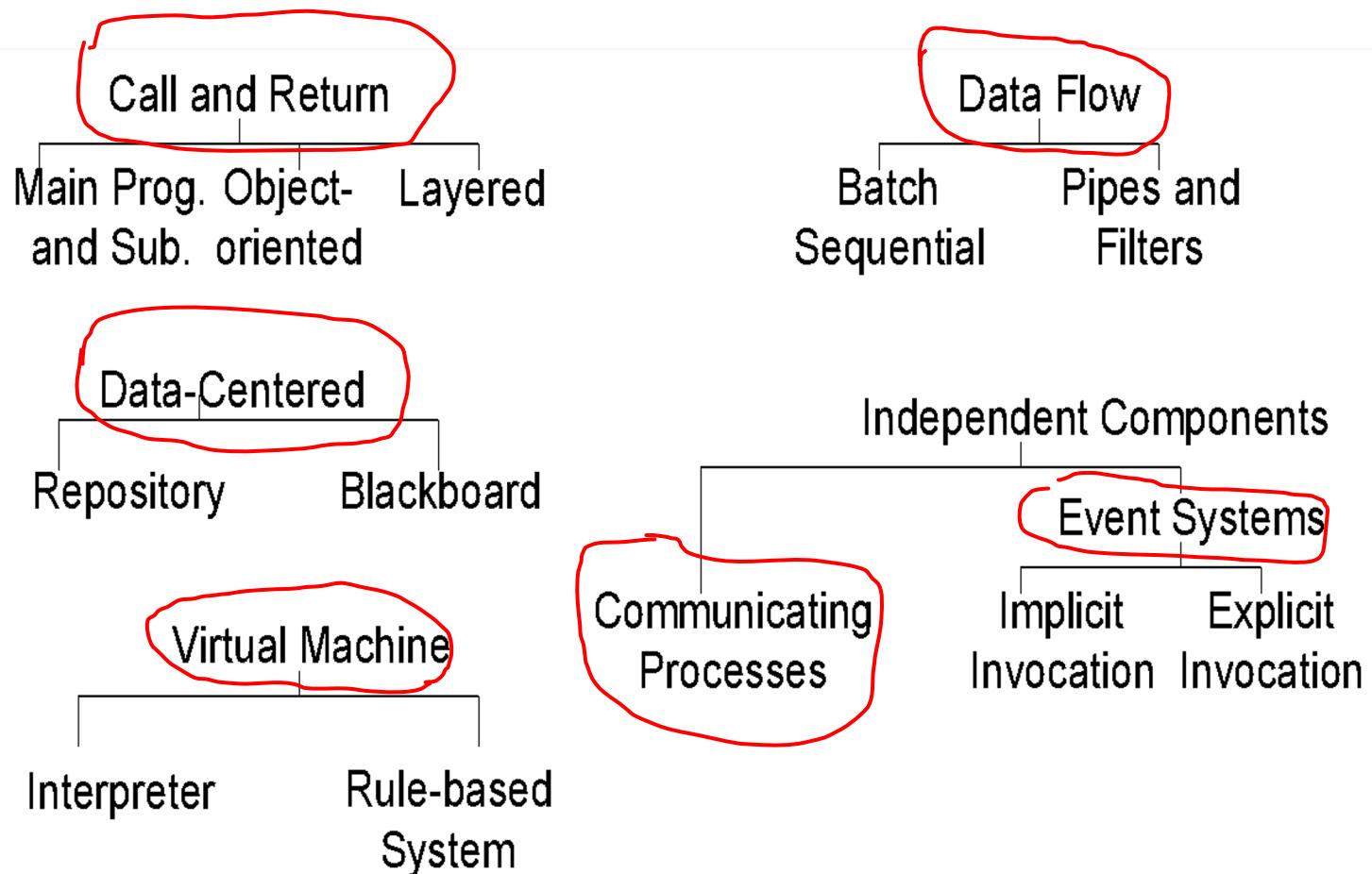
- Recurring (and proven) architectural design
- Definition of common vocabulary

Each style has:

- components, connectors, key characteristics, strengths and weaknesses, variants and specializations.

*From Chapter 5, Software Architecture in Practice, p. 94*

# A catalog of architectural styles



From Chapter 5, Software Architecture in Practice, p. 95

# Call-and-Return style

## Goal:

- achieve modifiability and scalability

## Substyles:

- **Main-program-and-subroutine**

Decompose program hierarchically. Each component gets control and data from its parent.

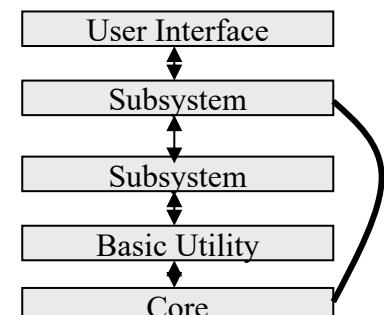
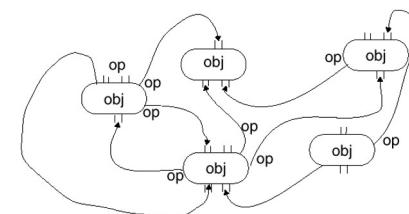
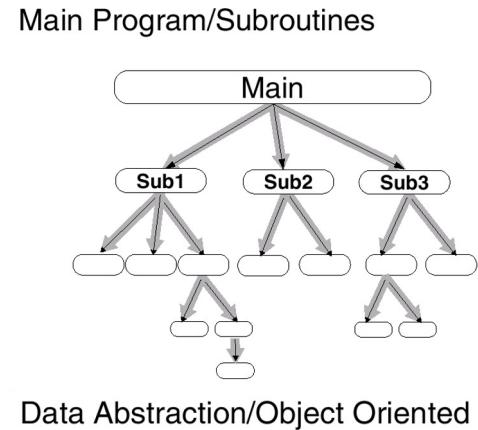
- Data Abstraction / Object-oriented

Achieve modifiability by encapsulating internal secrets from environment. Access to objects is only through methods. Object-oriented paradigm is distinguished from ADT by inheritance and polymorphism

- Layered

Seeks modifiability and portability. Optimally, each layer communicates only with its neighbors.

Sometimes, must layer bridge for performance:  
decreases benefits of style



# Data-Flow style

Goals: achieve reuse and modifiability

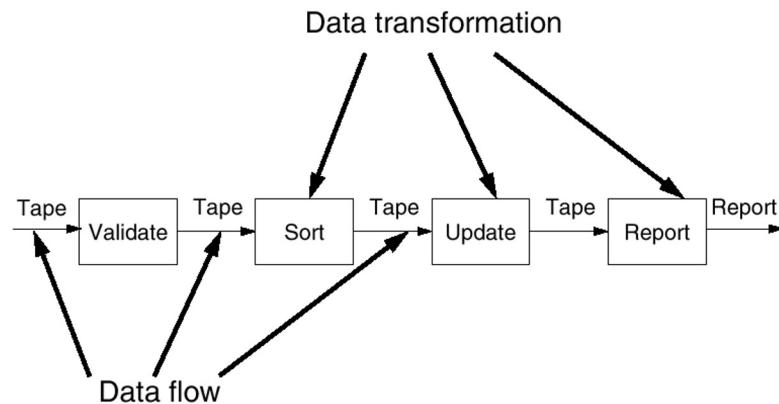
Components are independent programs

- each step runs to completion before the next starts
- each batch of data is transmitted as a whole between steps

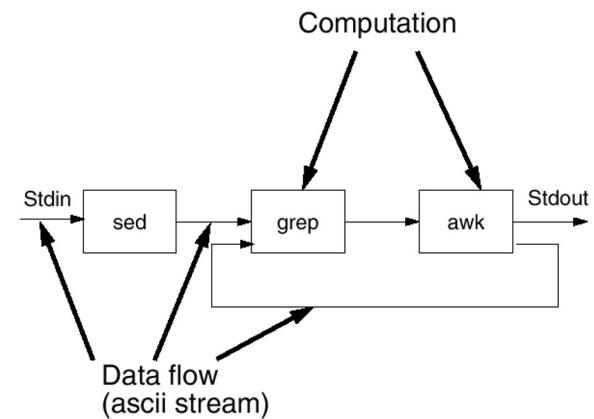
Classic data-processing approach

Substyles: **Batch sequential; Pipes and filters.**

Batch sequential



Pipes and Filters



# Data-Centered style

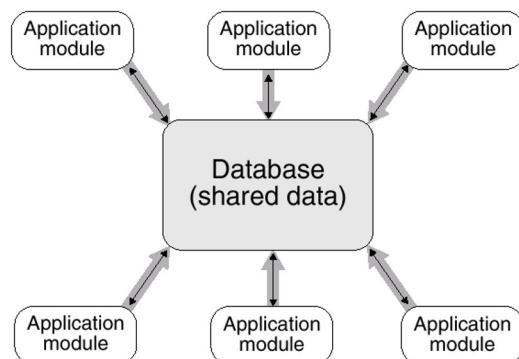
Two-substyles: Repository and Blackboard

When a system can be described as a centralized data store that communicates with a number of clients

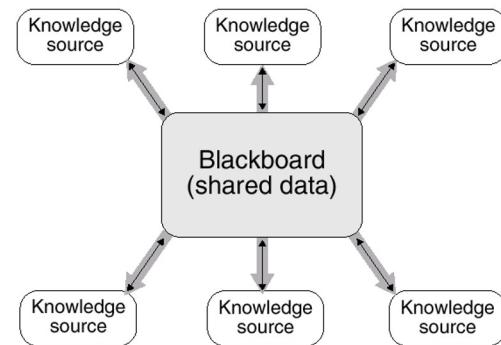
In a (passive) repository, such as shown on left, data might be stored in a file.

In an active repository, such as a blackboard, the blackboard notifies clients when data of interest changes (so there would be control from data to clients)

Database



Blackboard



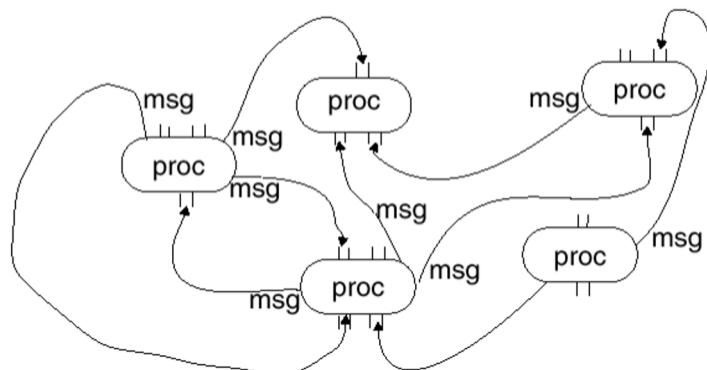
# Independent components style

Goals: achieve modifiability by decoupling various parts of the computation

Approach: have independent processes or objects communicate through messages

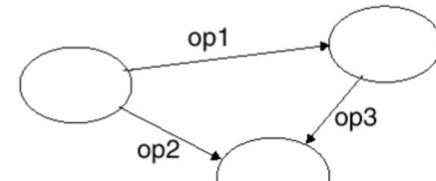
Substyles: **Communicating processes; Event systems.**

## Communicating Processes

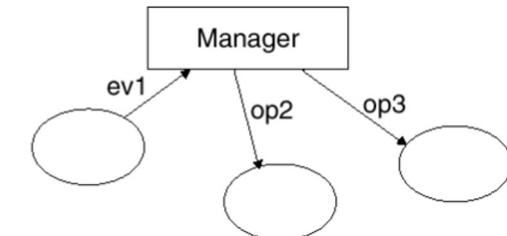


## Event systems with Implicit vs. Explicit Invocation

Explicit

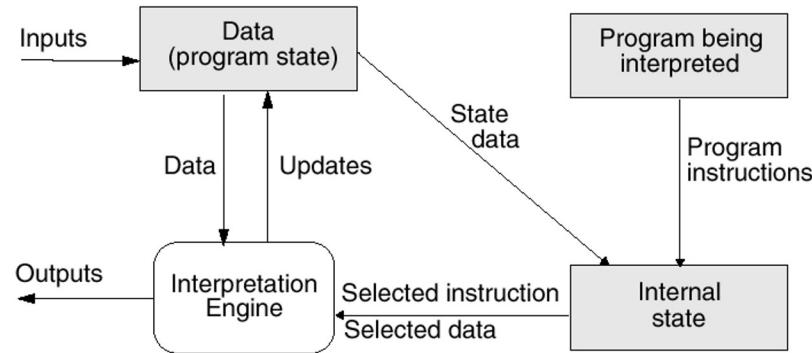


Implicit

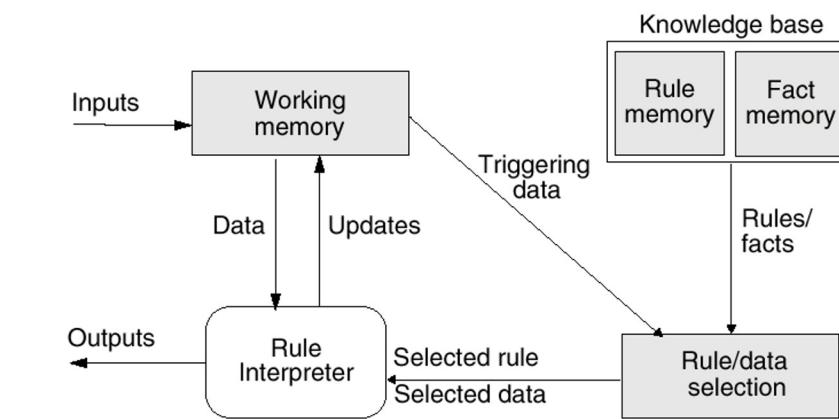


# Virtual machine style

## Interpreter



## Rule-Based System



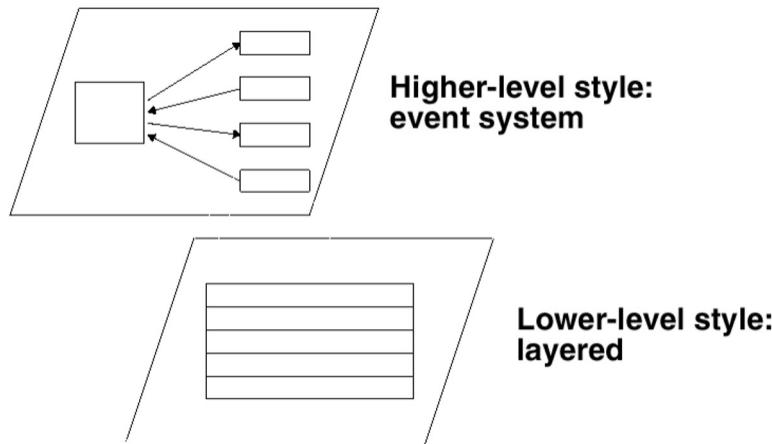
# Mixing styles

Systems are generally built from multiple styles

Three kinds of heterogeneity:

- *Locationally heterogeneous*: run-time structures reveal different styles in different areas
- *Hierarchically heterogeneous*: a component of one style, when decomposed is structured according to another style
- *Simultaneously heterogeneous*: different styles depending on point of view

Example



# Characterizing and comparing styles

The following categories are useful in comparing and characterizing styles:

- What kinds of components and connectors are used in the style?
- What are the control structures?
- How is data communicated?
- How do data and control interact?
- What kind of reasoning does the style support?

# Components & Connectors

*“A component is a unit of software that performs some function at runtime” [p.105]*

*“A connector is a mechanism that mediates communication, coordination, or cooperation among components” [p. 105]*

# Control issues

How does control pass among components?

*Topology:*

- What is the topology of a batch-sequential data-flow style?
- What is the topology of a main-program-and-subroutine style?

*Synchronicity:* How interdependent are the component's actions upon each other's control states?

- E.g., lockstep - state of one component implies state of all others
- E.g., synchronous - components synchronize regularly; other state relationships are unpredictable

*Binding Time:* When is the partner in a communication established?

- Program-write time?
- Compile-time?

# Issues...

## Data Issues

- *Topology*
- *Continuity*: Continuous vs. sporadic; volume
- *Mode*: Passed, shared, copy-out-copy-in, etc.
- *Binding Time*: Same as control issue

## Control/Data Interaction Issues

- *Shape* of control and data topologies
- *Directionality*: does data flow in direction of control?

## Type of Reasoning

# Group work

Split into groups

Identify the main architectural styles of a well-known system:

- Identify the components and connectors of those styles;
  - What kinds of components and connectors are used in the style?
  - What are the control structures?
  - How is data communicated?
  - How do data and control interact?
  - What kind of reasoning does the style support?
- Systems can be heterogeneous in terms of styles, mainly three kinds:
  - *Locationally* heterogeneous: run-time structures reveal different styles in different areas
  - *Hierarchically* heterogeneous: a component of one style, when decomposed is structured according to another style
  - *Simultaneously* heterogeneous: different styles depending on point of view
- Your system may not be "unifiable" into a single responsibility. In this case, divide it into orthogonal subsystems, and treat each other separately:
  - But how will they integrate/communicate?

Identify architectural / design problems

10 seconds presentation of the results

system name, style name 1, style name 2(, style name 3)

# **Thank you!**

# **Software Architecture**

FEUP-M.EIC-ASSO-2024

**Ademar Aguiar, Neil Harrison**

# POSA Patterns: the books

A System of Patterns. 1996. POSA1

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal

Patterns for Concurrent and Networked Objects. 2000.  
POSA2

Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann

Patterns for Resource Management. 2007. POSA3

Michael Kircher, Prashant Jain

A Pattern Language for Distributed Computing. 2007.  
POSA4

Frank Buschmann, Kevlin Henney, Douglas C. Schmidt

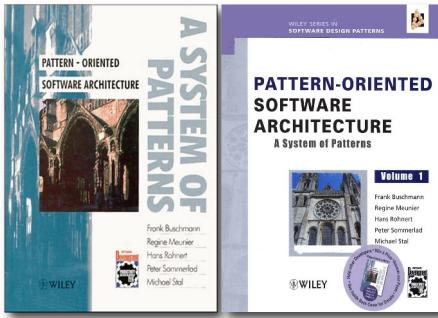
On Patterns and Pattern Languages. 2007. POSA5

Frank Buschmann, Kevlin Henney, Douglas C. Schmidt



# *Pattern Oriented Software Architecture: A System of Patterns*

(POSA 1)



Frank Buschmann  
Regine Meunier  
Hans Rohnert  
Peter Sonnenfeld  
Michael Stal

Wiley, 1996

# **Pattern Oriented Software Architecture**

## **A System of Patterns, Buschmann et al, 1996**

A book about patterns for software architecture.

A book to support both novices and experts in software development.

It should support experts in the design of large-scale and complex software systems with defined properties.

It should also enable them to learn from the experience of other experts.

# Architectural Patterns

## Definition (Buschmann et al)

- An **architectural pattern** expresses a fundamental structural organization schema for software systems. It provides a set of predefined **subsystems**, specifies their **responsibilities**, and includes rules and guidelines for organizing the **relationships** between them.

## Architectural templates

- Architectural patterns are templates for concrete software architectures.
- They specify the system-wide structural properties of an application, and have an impact on the architecture of its subsystems.

## Fundamental design decisions

- The selection of an architectural pattern is therefore a fundamental design decision when developing a software system.

# Four Categories of Architectural Patterns (POSA 1)

## From Mud to Structure

- To help avoiding a 'sea' of components or objects by supporting a controlled decomposition of a task into cooperating subtasks.
- Patterns: **Layers** (31), **Pipes and Filters** (53), **Blackboard** (71).

## Distributed Systems

- To help on architecting distributed systems.
- Patterns: **Broker** (99); refers to **Microkernel** (171), **Pipes and Filters** (53).

## Interactive Systems

- To support the structuring of software systems that feature HCI.
- Patterns: **Model-View-Controller** (125), **Presentation-Abstraction-Control** pattern (145).

## Adaptable Systems

- To support extension of applications and their adaptation to evolving technology and changing functional requirements.
- Patterns: **Reflection** (193), **Microkernel** (171).

# Design Patterns

Definition (Gamma et al.)

- A design pattern provides a scheme for **refining the subsystems or components** of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

Design patterns are medium-scale patterns

- **Smaller in scale** than architectural patterns
- Tend to be **independent of a particular programming language or programming paradigm**.
- The application of a design pattern has **no effect on the fundamental structure** of a software system, but may have a strong influence on the architecture of a subsystem.
- Many design patterns provide **structures for decomposing** more complex services or components.
- Others address the effective **cooperation** between them.

# Categories of Design Patterns (POSA1)

## Structural Decomposition

- To support a suitable decomposition of subsystems and complex components into cooperating parts.
- Patterns: **Whole-Part** (225)

## Organization of Work

- To define how components collaborate together to solve a complex problem.
- Patterns: **Master-Slave** (245)

## Access Control

- To guard and control access to services or components.
- Patterns: **Proxy** (263).

## Management

- To handle homogenous collections of objects, services and components in their entirety.
- Patterns: **Command Processor** (277), **View Handler** (291).

## Communication

- To organize communication between components.
- Patterns: **Forwarder-Receiver** (307), **Client Dispatcher-Server** (323), **Publisher-Subscriber** (339).

# Pattern form

**Name** The name and a short summary of the pattern.

**Also Known As** Other names for the pattern, if any are known.

**Example** A real-world example demonstrating the problem and the pattern's need.

**Context** The situations in which the pattern may apply

**Problem** The problem the pattern addresses, including a discussion of its forces.

**Solution** The fundamental solution principle underlying the pattern.

**Structure** A detailed specification of the structural aspects of the pattern.

**Dynamics** Typical scenarios describing the run-time behavior of the pattern.

**Implementation** Guidelines or suggestions for implementing the pattern.

**Example Resolved** Discussion of any important aspects for resolving the example.

**Variants** A brief description of variants or specializations of a pattern.

**Known Uses** Examples of the use of the pattern, taken from existing systems.

**Consequences** The benefits the pattern provides, and any potential liabilities.

**See Also** References to patterns that solve similar problems, or that help refine it.

# Basic and Simple Patterns first

Simple patterns are easy to understand and appear in many well-structured software systems.

Architectural patterns

- [Layers \(31\)](#)
- [Pipes and Filters \(53\)](#)

Design patterns

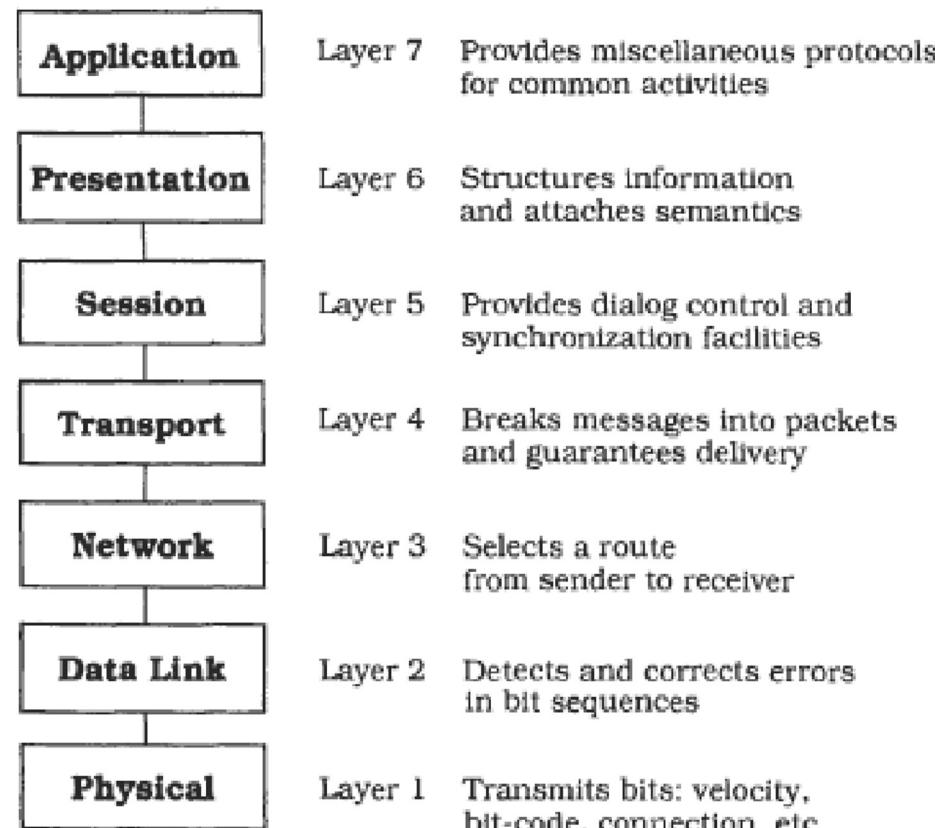
- [Proxy \(263\)](#)
- [Forwarder-Receiver \(307\)](#)

# Layers (31)

“The LAYERS pattern helps to **structure** applications that can be **decomposed** into **groups of subtasks** in which each group of subtasks is at a particular level of abstraction, granularity, hardware-distance, or other partitioning criteria”

# Layers (31) – Example

Example: networking protocols, OSI 7-layer model



# Layers (31) – Context, Problem

## Context

- A large system that requires decomposition

## Problem

- Imagine that you are designing a system whose dominant characteristic is a **mix of low- and high-level issues**, where **high-level operations rely on the lower-level ones**.  
(...)
- A typical pattern of communication flow consists of **requests moving from high to low level**, and **answers to requests**, incoming data or notification about events **traveling in the opposite direction**.  
(...)
- Such systems often also require some **horizontal structuring** that is orthogonal to their vertical subdivision. This is the case where several operations are on the same level of abstraction but are largely independent of each other.

# Layers (31) – Forces

**Late source code changes** should not ripple through the system. They should be confined to one component and not affect others.

**Interfaces should be stable**, and may even be prescribed by a standards body.

**Parts of the system should be exchangeable** without affecting the rest of the system. An extreme form of exchangeability might be a client component dynamically switching to a different implementation of a service that may not have been available at start-up.

**It may be necessary to build other systems at a later date** with the same low-level issues as the system you are currently designing.

Similar responsibilities should be grouped to help **understandability and maintainability**.

There is no 'standard' component granularity.

Complex components need **further decomposition**.

Crossing component boundaries may impede **performance**, for example when a substantial amount of data must be transferred over several boundaries, or where there are many boundaries to cross.

The system will be built by a **team of programmers**, and work has to be subdivided along clear boundaries.

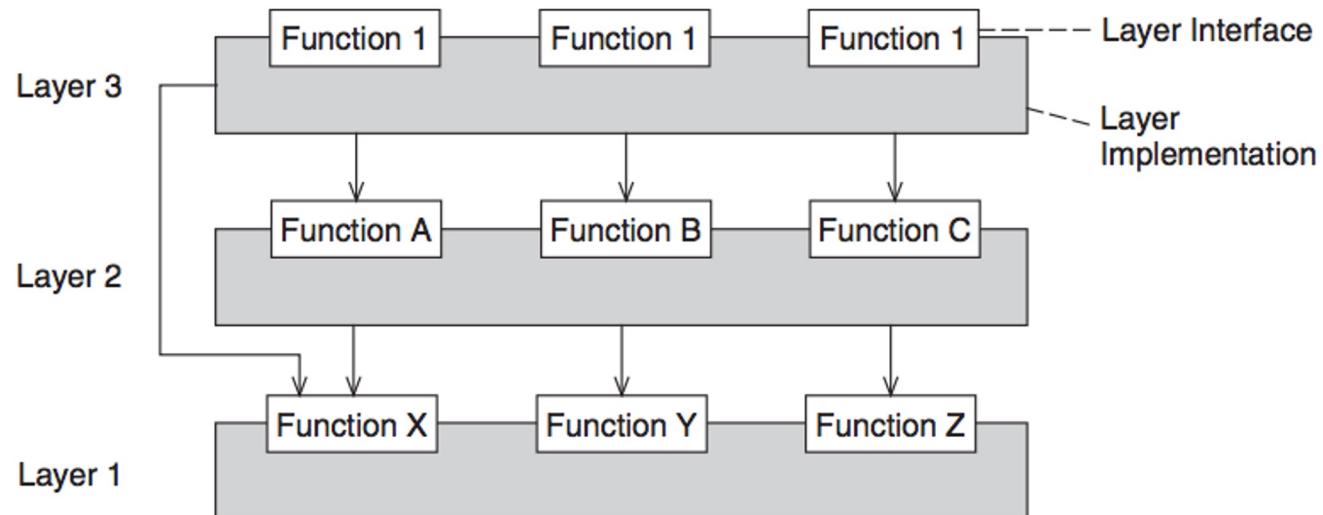
# Layers (31) – Solution

Structure your system into **an appropriate number of layers** and place them on top of each other.

**Start at the lowest level** of abstraction-call it Layer 1. This is the base of your system.

**Work your way up** the abstraction ladder by putting Layer J on top of Layer J - 1 **until you reach the top level** of functionality-call it Layer N.

# Layers – Structure



# Layers (31) - Dynamics

## Scenario I: “best-known”

- A client issues a request to Layer N. Since Layer N cannot carry out the request on its own, it calls the next Layer N - 1 for supporting subtasks

## Scenario II: “bottom-up communication”

- A chain of actions starts at Layer 1, and reports it to Layer 2.
- Top-down information and control flow are often described as **requests**.
- Bottom-up calls can be termed as **notifications**.

## Scenario III: “subset of layers”

- Situations where requests only travel through a subset of the layers.
- Examples: caching mechanisms.

## Scenario IV: “events stopped prematurely”

- An event is detected in Layer 1, but stops at Layer **3** instead of traveling all the way up to Layer N.

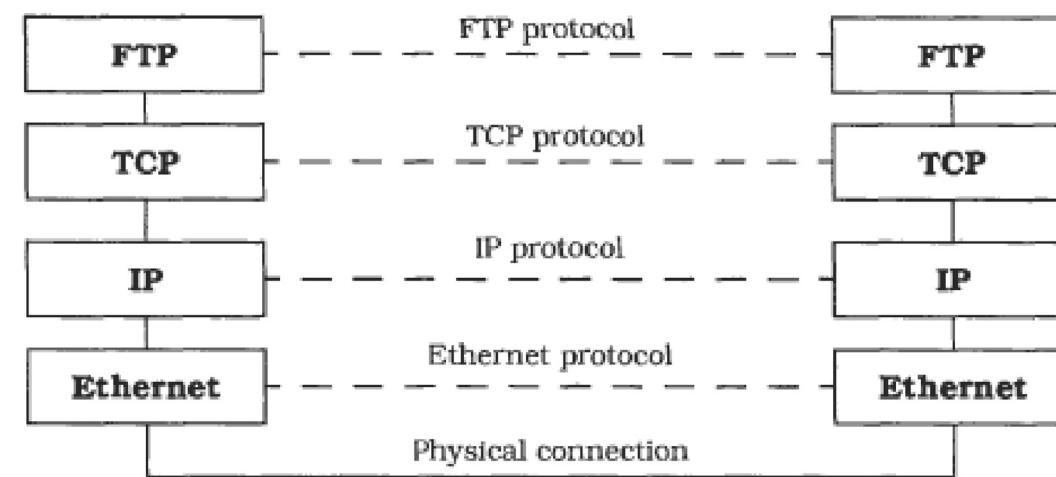
## Scenario V: “two stacks of N layers communicating with each other”

- Well-known scenario from communication protocols, where the stacks are known as protocol stacks.

# Layers (31) - Implementation

1. Define the abstraction criterion for grouping tasks into layers.
2. Determine the number of abstraction levels according to your abstraction criterion.
3. Name the layers and assign tasks to each of them.
4. Specify the services.
5. Refine the layering.
6. Specify an interface for each layer.
7. Structure individual layers.
8. Specify the communication between adjacent layers.
9. Decouple adjacent layers.
10. Design an error-handling strategy.

# Layers (31) – Example Resolved



# Layers (31) - Variants

## Relaxed Layered System

- A variant of the Layers pattern less restrictive about the relationship between layers.
- In a Relaxed Layered System each layer **may use the services of all layers below it**, not only of the next lower layer.
- A layer may also be partially opaque meaning that **some of its services are only visible to the next higher layer**, while others are visible to all higher layers.
- The **gain of flexibility and performance** in a Relaxed Layered System is paid for by a **loss of maintainability !!!**

## Layering Through Inheritance

- In this variant lower layers are implemented as base classes.
- An advantage of this scheme is that higher layers can modify lower-layer services according to their needs.
- A drawback is that such an inheritance relationship closely ties the higher layer to the lower layer.

# Layers (31) – Known Uses

## Virtual Machines

- Platform specific code (JVM)
- Bytecode instructions (compiled programs)
- Java programs (source code)

## APIs

- Operating system functionalities
- High-level functionalities
- Utility functionalities

## Information Systems

- Presentation
- Application logic
- Domain layer
- Database

# Layers (31) – Consequences

## Benefits

- Reuse of layers
- Support for standardization
- Dependencies are kept local
- Exchangeability

## Liabilities

- Cascades of changing behavior
- Lower efficiency
- Unnecessary work
- Difficulty of establishing the correct granularity of layers

# **Layers (31) – See also**

Composite message pattern

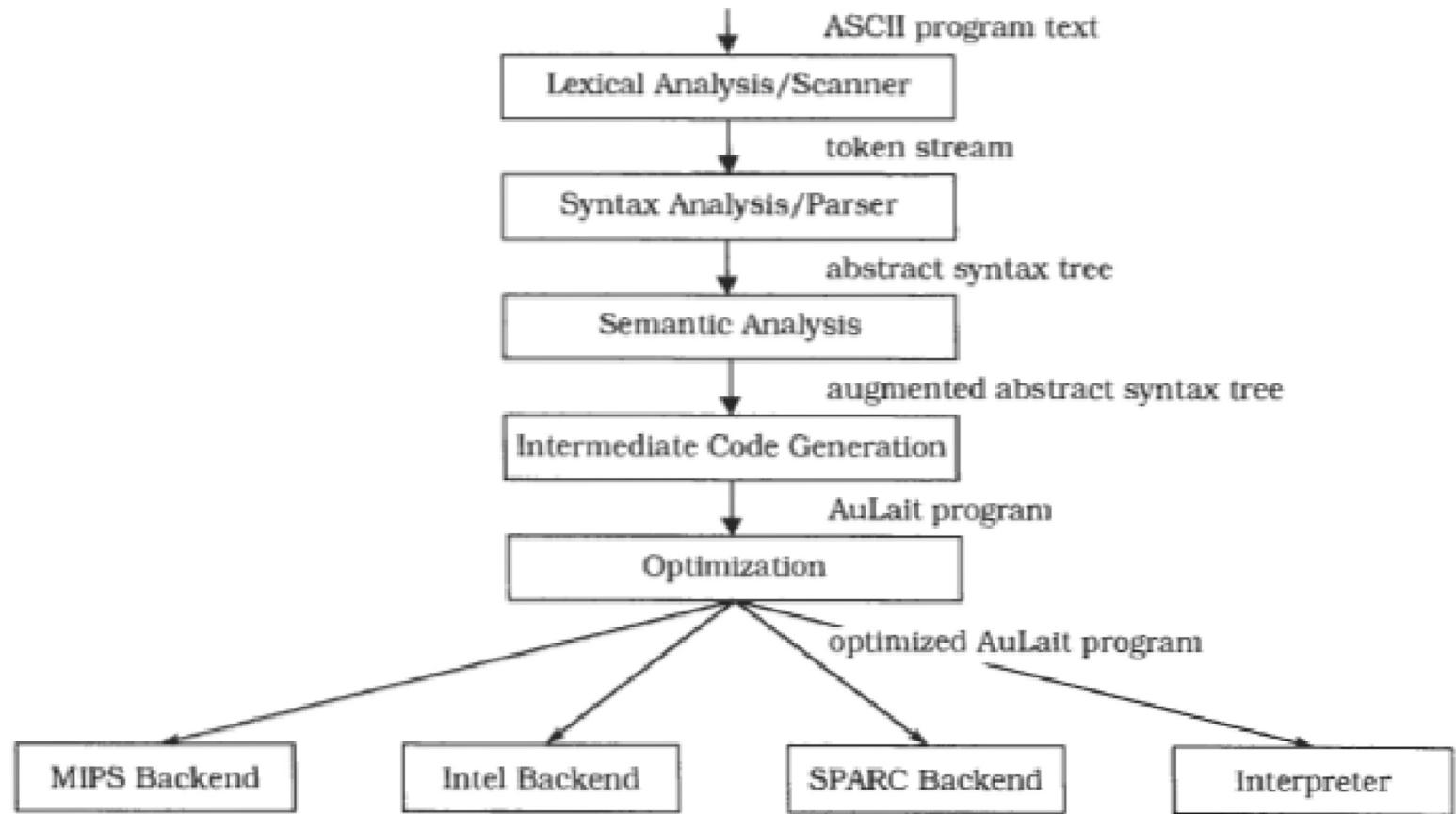
A Microkernel architecture (171)

The Presentation-Abstraction-Control architectural pattern (145)

## Pipes and Filters (53)

*“The Pipes and Filters architectural pattern provides a **structure** for systems that **process a stream of data**. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems”*

# Pipes and Filters (53) - Example



# Pipes and Filters (53) – Context, Problem

## Context

- Processing data streams.

## Problem

- Imagine you are building a system that must process or transform a stream of input data.
- Implementing such a system as a single component may not be feasible for several reasons:
  - the system has to be built by several developers,
  - the global system task decomposes naturally into several processing stages,
  - and the requirements are likely to change.
- You therefore plan for future flexibility by exchanging or reordering the processing steps.

# Pipes and Filters (53) - Forces

**Future system enhancements should be possible** by exchanging processing steps or by recombination of steps, even by users.

**Small processing steps are easier to reuse** in different contexts than large components.

**Non-adjacent processing steps do not share information.**

**Different sources of input data exist.**

It should be possible **to present or store results in various ways.**

**Explicit storage of intermediate results** for further processing in files clutters directories and **is error-prone**, if done by users.

You may not want to rule out **multi-processing** the steps, for example running them in parallel or quasi-parallel.

# Pipes and Filters (53) - Solution

The Pipes and Filters architectural pattern divides the task of a system into several **sequential processing steps**.

**These steps are connected by the data flow** through the system—the output data of a step is the input to the subsequent step.

Each processing step is implemented by a **filter** component.

**A filter consumes and delivers data incrementally** - in contrast to consuming all its input before producing any output—to achieve low latency and enable real parallel processing.

The input to the system is provided by a data source such as a text file.

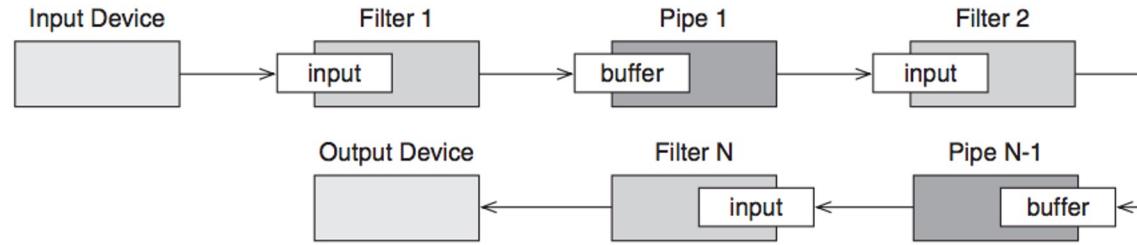
The output flows into a data sink such as a file, terminal, animation program and so on.

The data source, the filters and the data sink are connected sequentially by **pipes**.

Each pipe implements the data flow between adjacent processing steps.

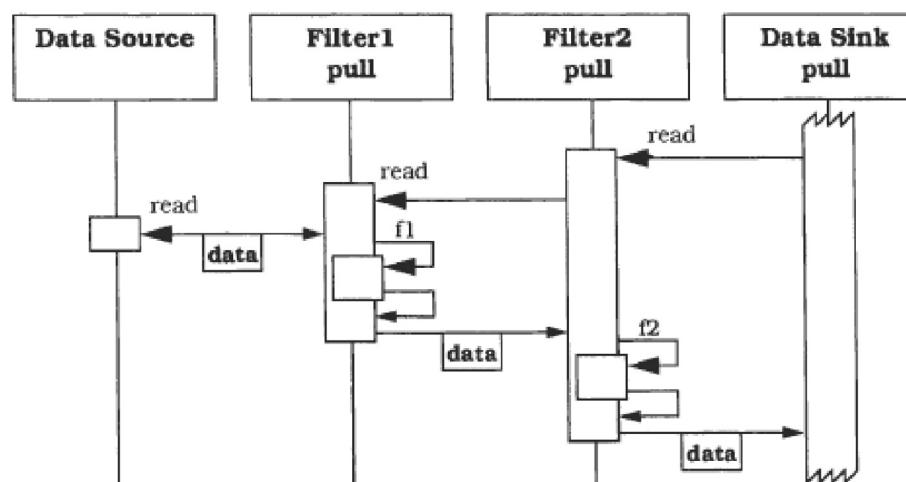
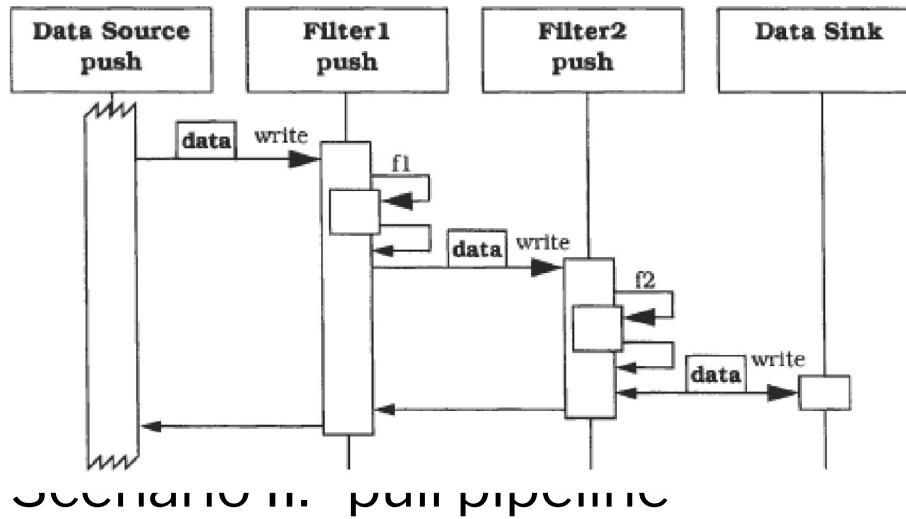
The sequence of filters combined by pipes is called a **processing pipeline**.

# Pipes and Filters – Structure



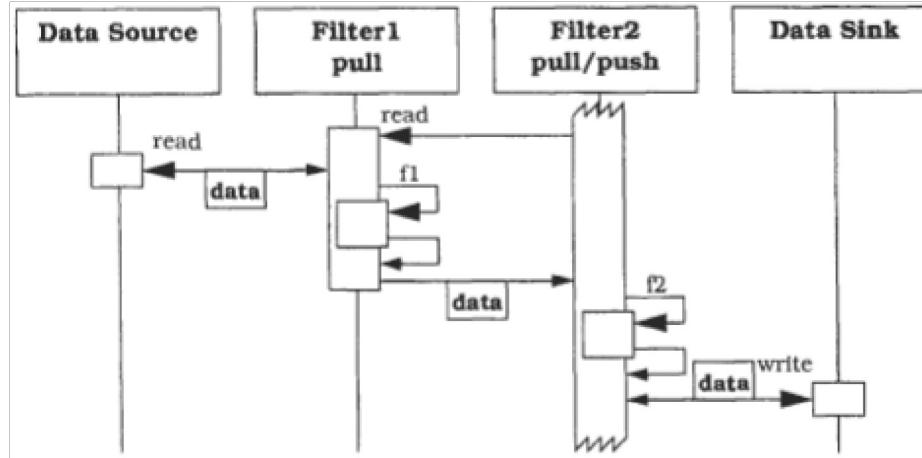
# Pipes and Filters (53) - Dynamics

Scenario I: “push pipeline”

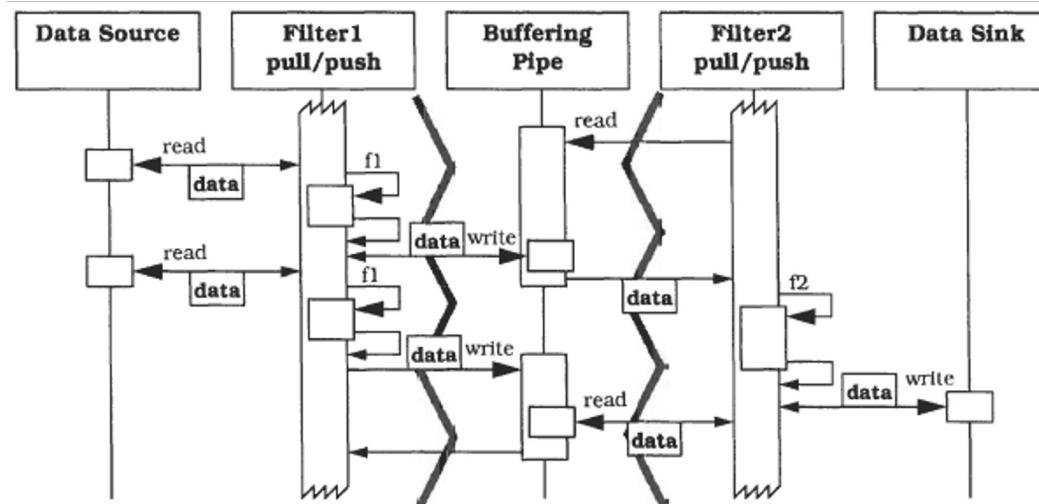


# Pipes and Filters (53) - Dynamics

Scenario III: “push-pull pipeline”



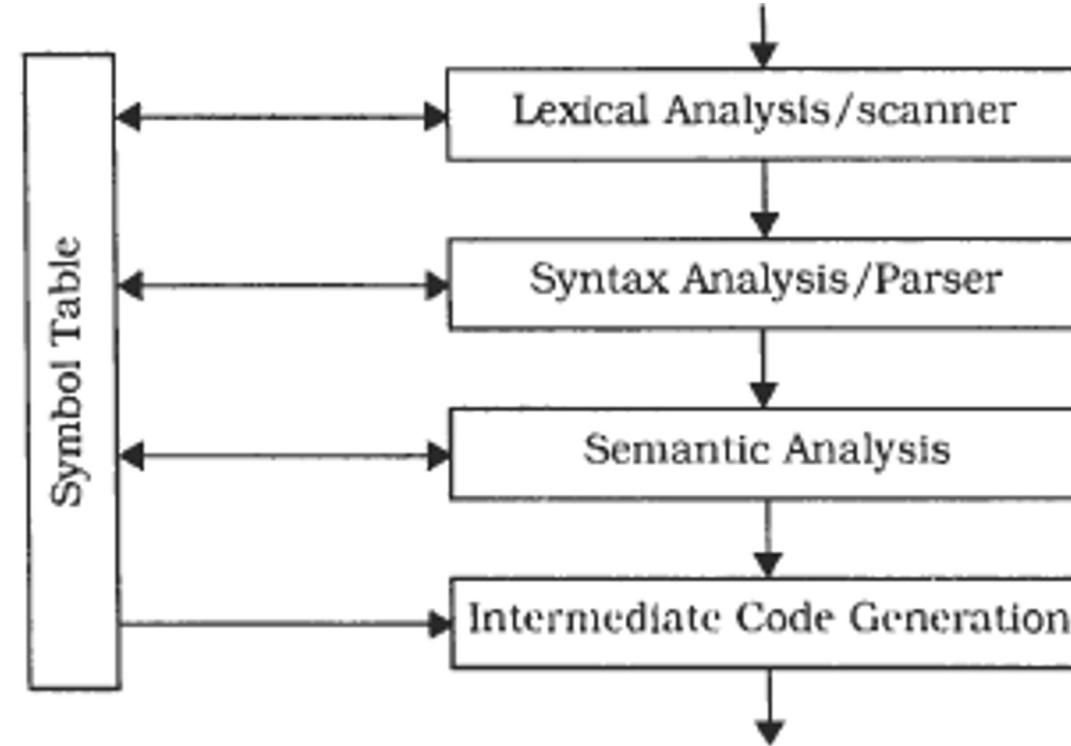
Scenario IV: “loop push-pull pipeline”



# Pipes and Filters (53) – Implementation

1. Divide the system's task into a sequence of processing stages.
2. Define the data format to be passed along each pipe.
3. Decide how to implement each pipe connection.
4. Design and implement the filters.
5. Design the error handling.
6. Set up the processing pipeline.

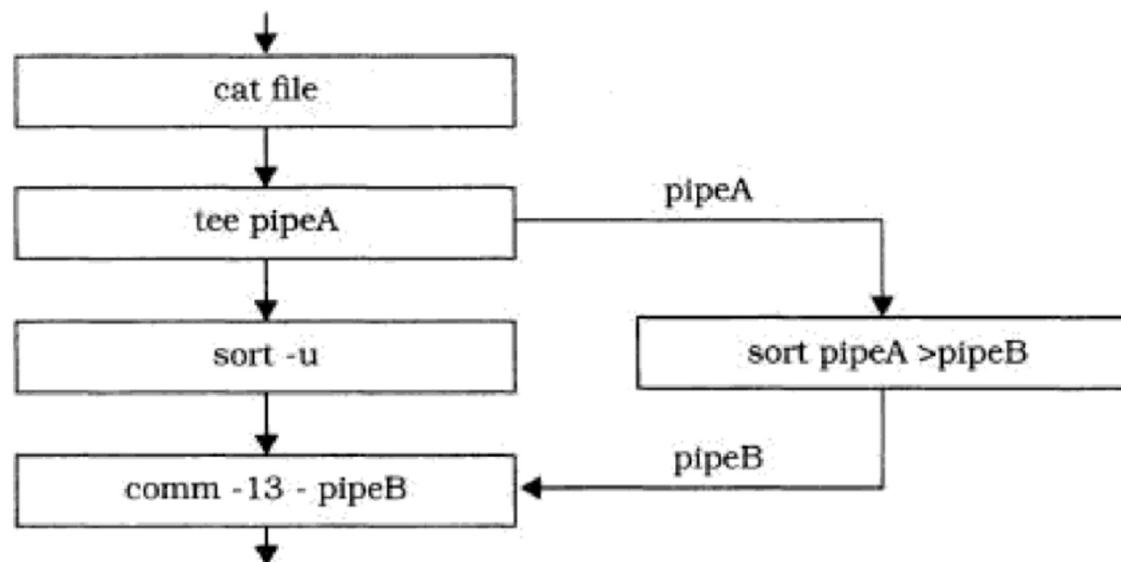
# Pipes and Filters (53) – Example Resolved



# Pipes and Filters (53) – Variants

## Tee and Join pipeline

```
# first create two auxiliary named pipes to be used  
mknod pipeA p  
mknod pipeB p  
# now do the processing using available UNIX filters  
# start side fork of processing in background:  
sort pipeA > pipeB &  
# the main pipeline  
cat file | tee pipeA | sort -u | comm -13 - pipeB
```



# Pipes and Filters (53) – Known Uses

UNIX

CMS pipelines

LASSPTools

# Pipes and Filters (53) – Consequences

## Benefits

- No intermediate files necessary, but possible
- Flexibility by filter exchange
- Flexibility by recombination
- Reuse of filter components
- Rapid prototyping of pipelines
- Efficiency by parallel processing

## Liabilities

- Sharing state information is expensive or inflexible
- Efficiency gain by parallel processing is often an illusion (data transfer costs, filters consuming all the input at once, context-switching costs, filter synchronization difficulties)
- Data transformation overhead
- Error handling (the Achilles' heel...)

# Pipes and Filters (53) – See also

## Layers (31)

- The **Layers** pattern (31) is better suited to systems that require reliable operation, because it is easier to implement error handling than with Pipes and Filters.
- However, Layers lacks support for the easy recombination and reuse of components that is the key feature of the Pipes and Filter pattern.

# Team work

Review your architecture of the Library System

- Revisit the several **subsystems** and how they are **connected**
- What are the main **architectural and design challenges**?
- Which **patterns** can you use to address those challenges?

# Reference

Pattern Oriented Software Architecture, A System of Patterns, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sornmerlad, Michael Stal, Wiley, 1996.

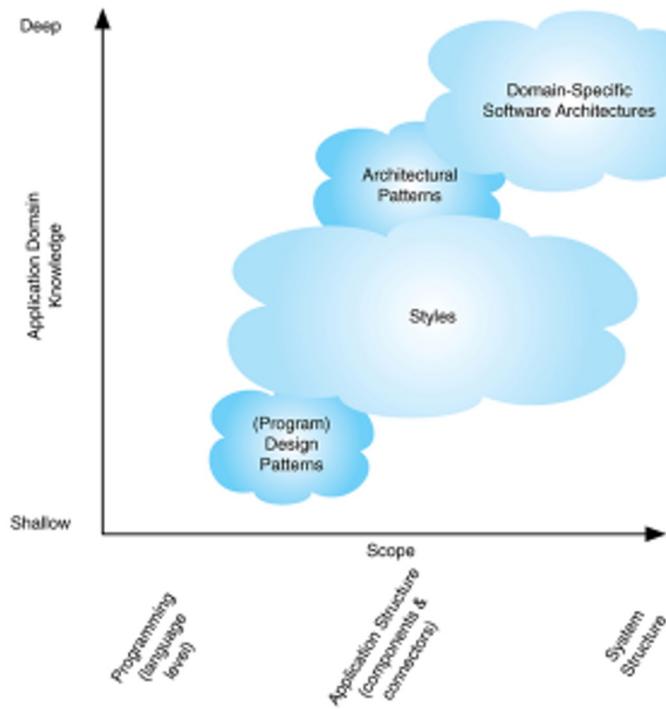
# **Thank you!**

# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**

# Patterns, Styles, and DSSAs



Fig\_04\_02

# A Gallery of Patterns

Some of the most common architecture patterns

# Three Tier

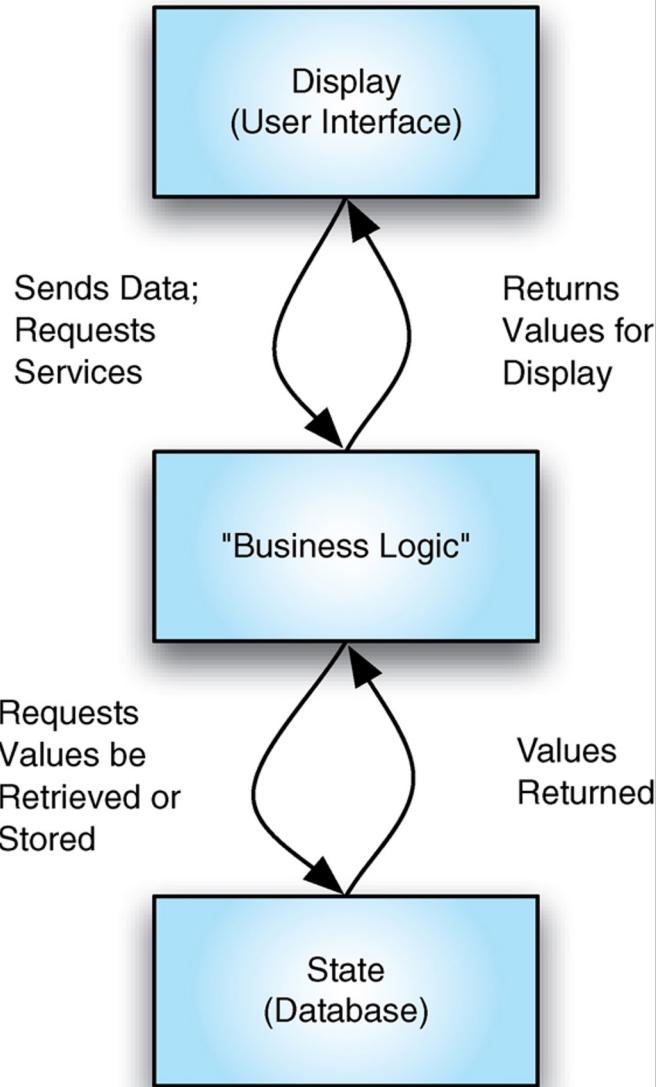
AKA: state-logic-display

User Interface

Business Logic

Data store (state)

Special case of the Layers Pattern



# Batch Sequential

The whole task is subdivided into small processing steps.

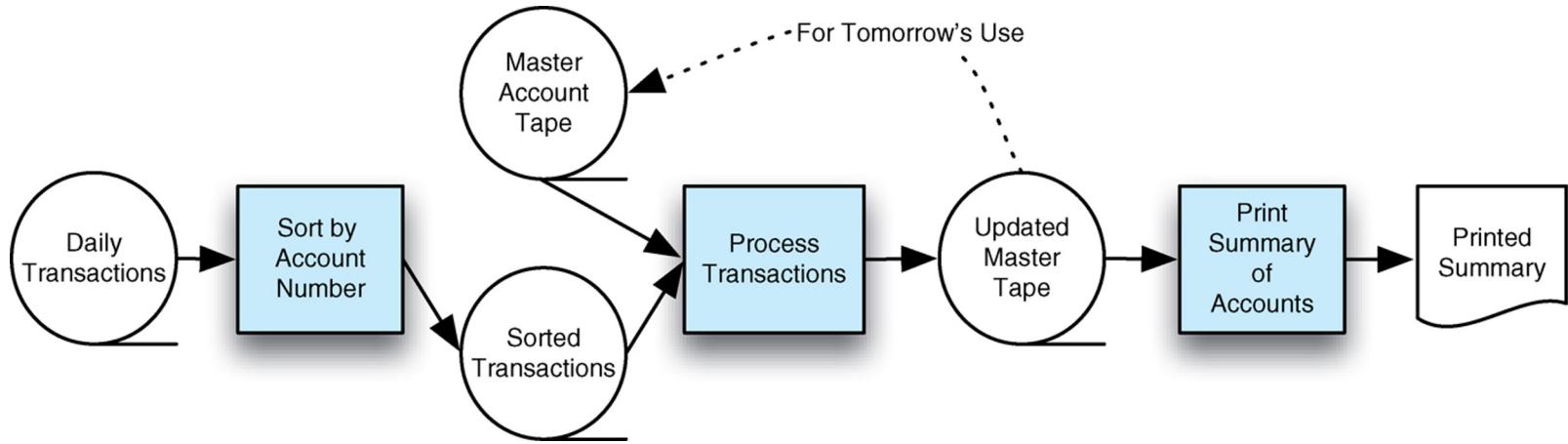
Each step is realized as a separate independent component.

Each step runs to completion and calls the next sequential step until the whole task is fulfilled.

During each step a batch of data is processed and sent as a whole to the next step.

Early computers were batch machines.

# Batch Sequential



Financial records processed in batch-sequential architecture

# When to use Pipes and Filters (From MS Azure)

Use this pattern when:

The processing required by an application can easily be broken down into a set of **independent** steps.

The processing steps performed by an application have **different scalability requirements**.

It's possible **to group filters that should scale together** in the same process. For more information, see the Compute Resource Consolidation pattern.

**Flexibility** is required to allow reordering of the processing steps performed by an application, or the capability to add and remove steps.

The system can benefit from **distributing the processing for steps across different servers**.

A reliable solution is required that **minimizes the effects of failure in a step while data is being processed**.

This pattern might not be useful when:

The processing steps performed by an application **aren't independent**, or they must be performed together as part of the same transaction.

The amount of **context or state information required by a step makes this approach inefficient**. It might be possible to persist state information to a database instead, but don't use this strategy if the additional load on the database causes excessive contention.

# What is the difference between Batch-Sequential and Pipes and Filters?

In Batch-Sequential, each step finishes processing the set of data before the next step begins

In Pipes and Filters, the next step may begin as soon as the previous step begins outputting data.

Can you think of applications which are appropriate for either?

Compiler: which is it?

Processing a data stream: which is it?



# Sense-Compute-Control

Domain: embedded control applications

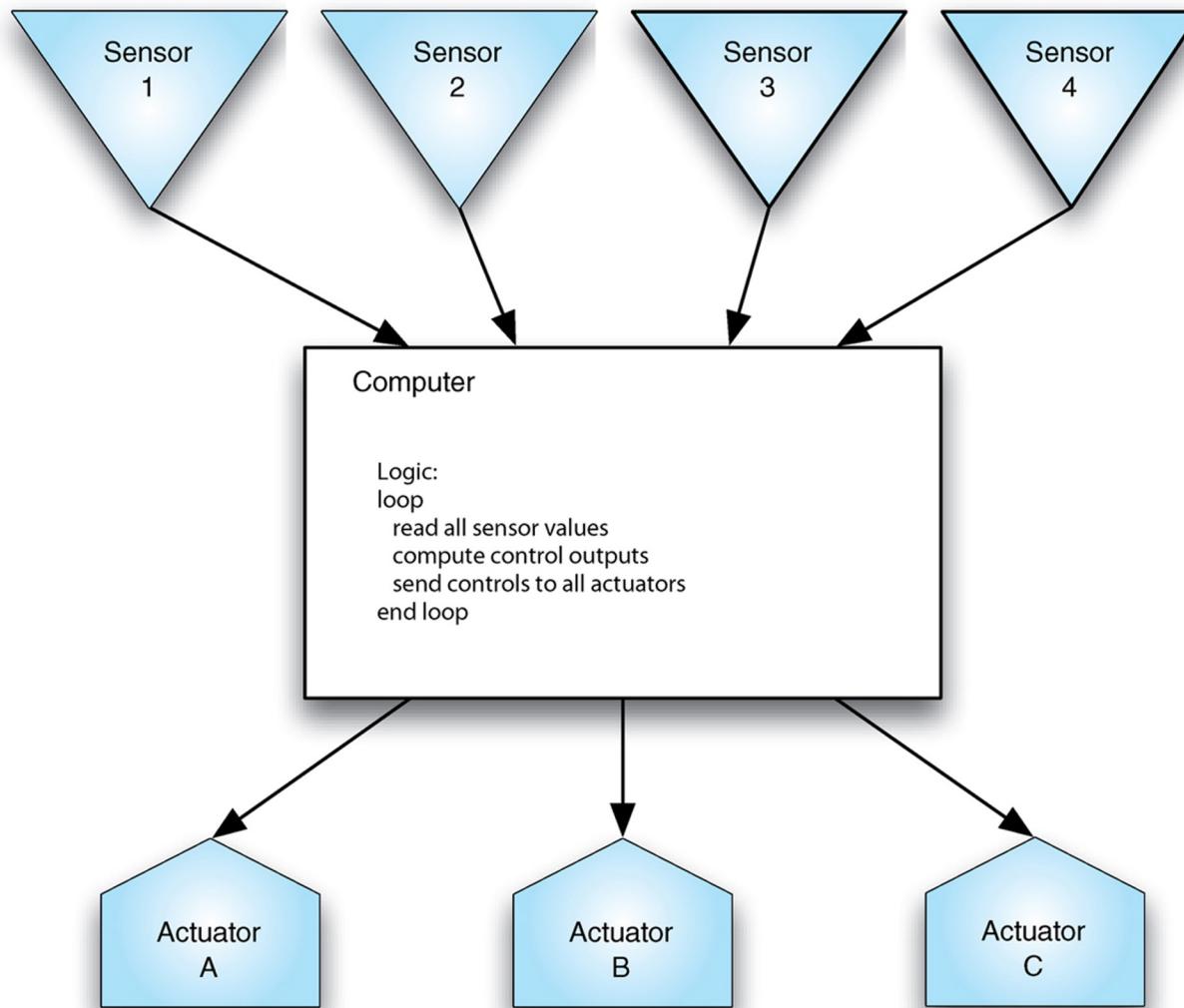
Read sensors, send actions to actuators

Similar to Event-Driven pattern

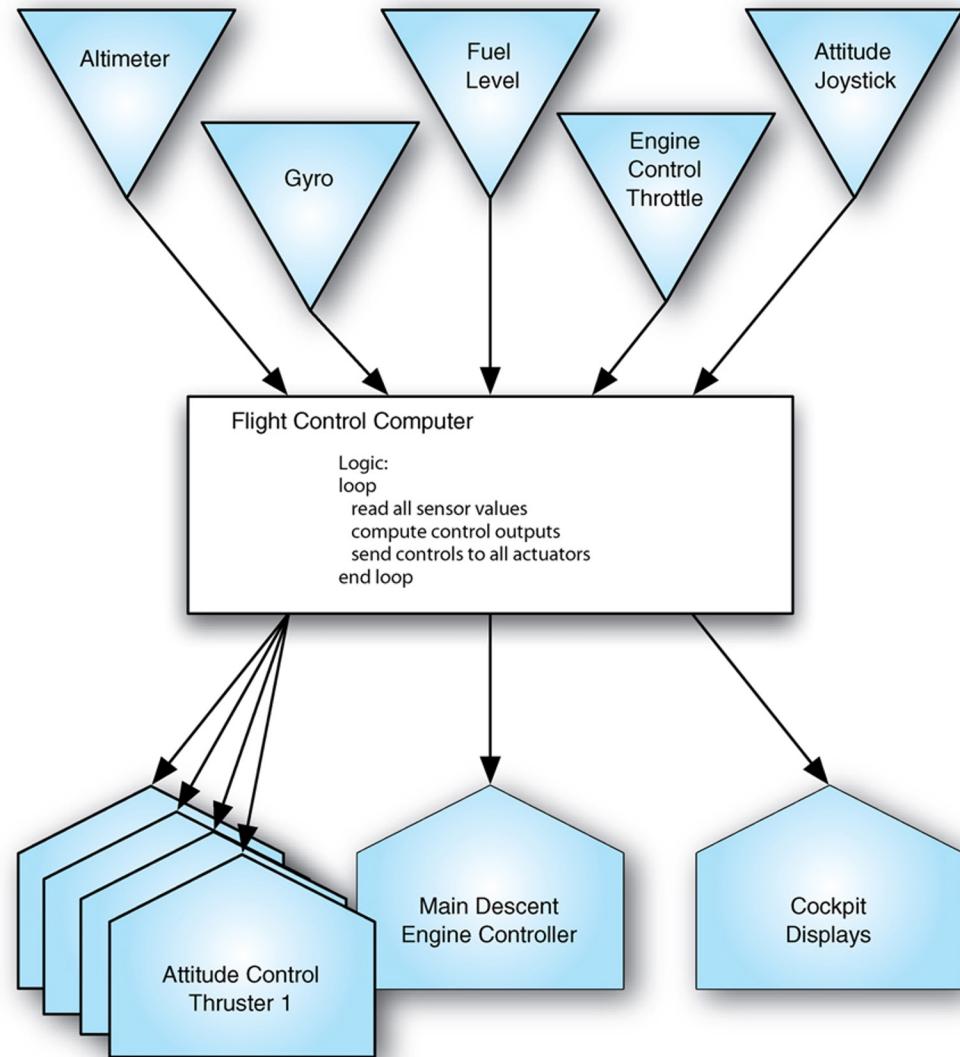
Difference:

- Event driven: handling events, rather than polling

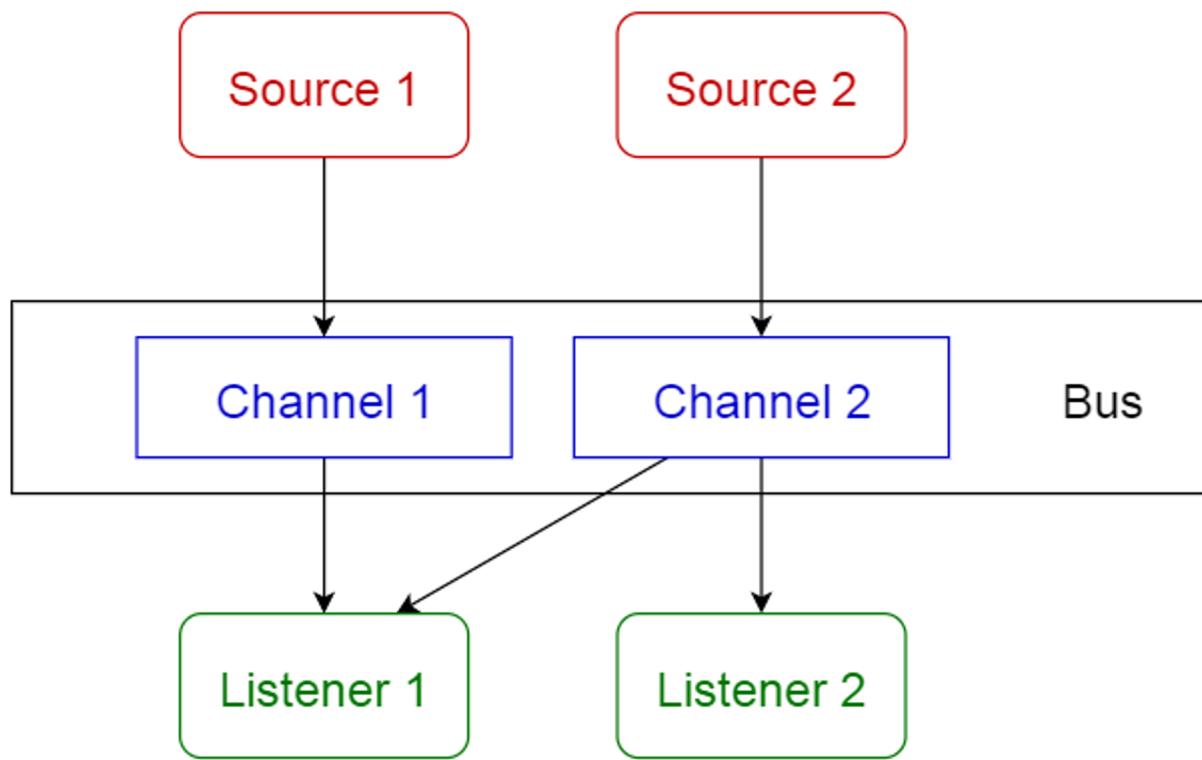
# Sense-Compute-Control



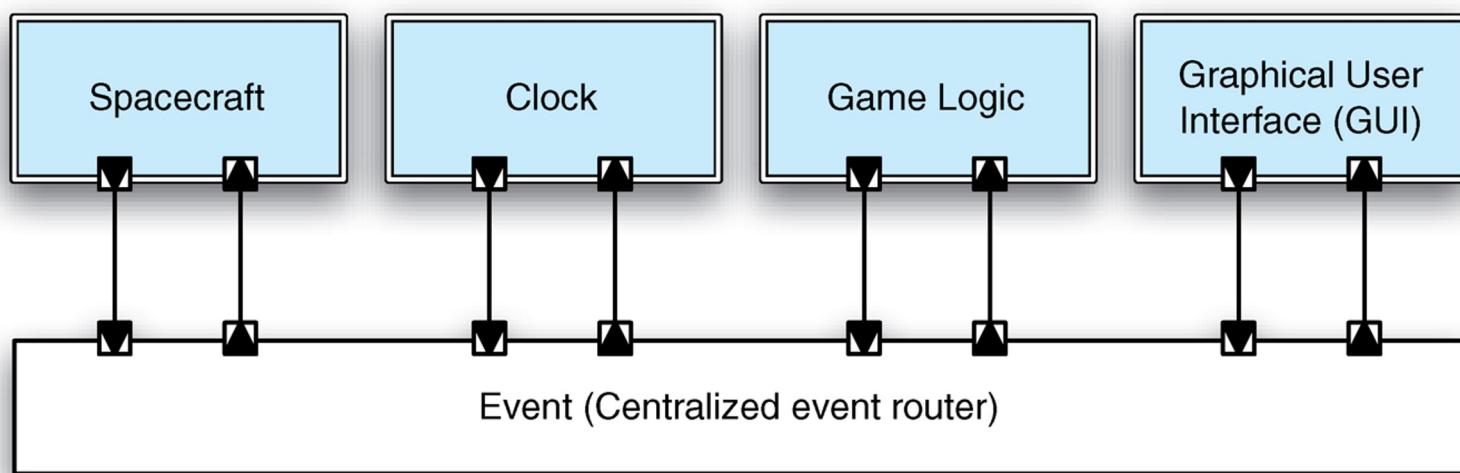
# Sense-Compute-Control, Lunar Lander



# Event-Driven



# Event-based System



# What is an Event-Driven Architecture? (From AWS)

An event-driven architecture uses events to trigger and communicate between decoupled services and is common in modern applications built with microservices. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-commerce website. Events can either carry the state (the item purchased, its price, and a delivery address) or events can be identifiers (a notification that an order was shipped).

Event-driven architectures have three key components: event producers, event routers, and event consumers. A producer publishes an event to the router, which filters and pushes the events to consumers. Producer services and consumer services are decoupled, which allows them to be scaled, updated, and deployed independently.

# Benefits of an event-driven architecture (from AWS)

## Scale and fail independently

By decoupling your services, they are only aware of the event router, not each other. This means that your services are interoperable, but if one service has a failure, the rest will keep running. The event router acts as an elastic buffer that will accommodate surges in workloads.

## Develop with agility

You no longer need to write custom code to poll, filter, and route events; the event router will automatically filter and push events to consumers. The router also removes the need for heavy coordination between producer and consumer services, speeding up your development process.

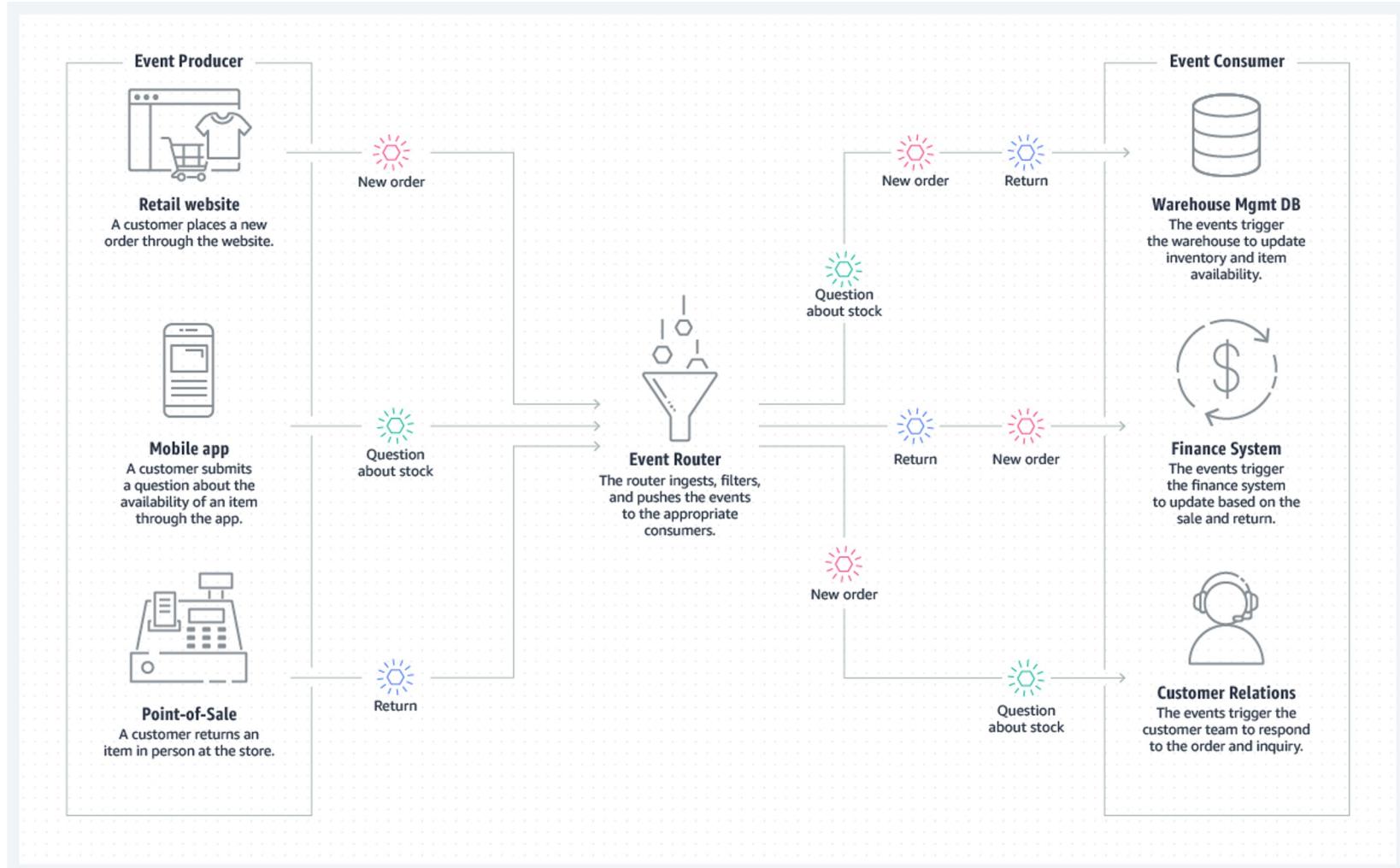
## Audit with ease

An event router acts as a centralized location to audit your application and define policies. These policies can restrict who can publish and subscribe to a router and control which users and resources have permission to access your data. You can also encrypt your events both in transit and at rest.

## Cut costs

Event-driven architectures are push-based, so everything happens on-demand as the event presents itself in the router. This way, you're not paying for continuous polling to check for an event. This means less network bandwidth consumption, less CPU utilization, less idle fleet capacity, and less SSL/TLS handshakes.

# Event driven architecture from AWS (caveat: also a Broker)



# Java Swing is event-driven

```
public class FooPanel extends JPanel implements ActionListener {  
    public FooPanel() {  
        super();  
  
        JButton btn = new JButton("Click Me!");  
        btn.addActionListener(this);  
  
        this.add(btn);  
    }  
  
    @Override  
    public void actionPerformed(ActionEvent ae) {  
        System.out.println("Button has been clicked!");  
    }  
}
```

# Microkernel

Provide a common plug-and-play interface for (usually) low level operations

Implements services that all systems in the “family” need

Accessed through APIs provided

Often a layer

Example: the JVM

Highlights:

- Portability!
- Also provides a common point for enhancements
- But performance may suffer
  - Example: common virtual machine for programming apps for mobile devices

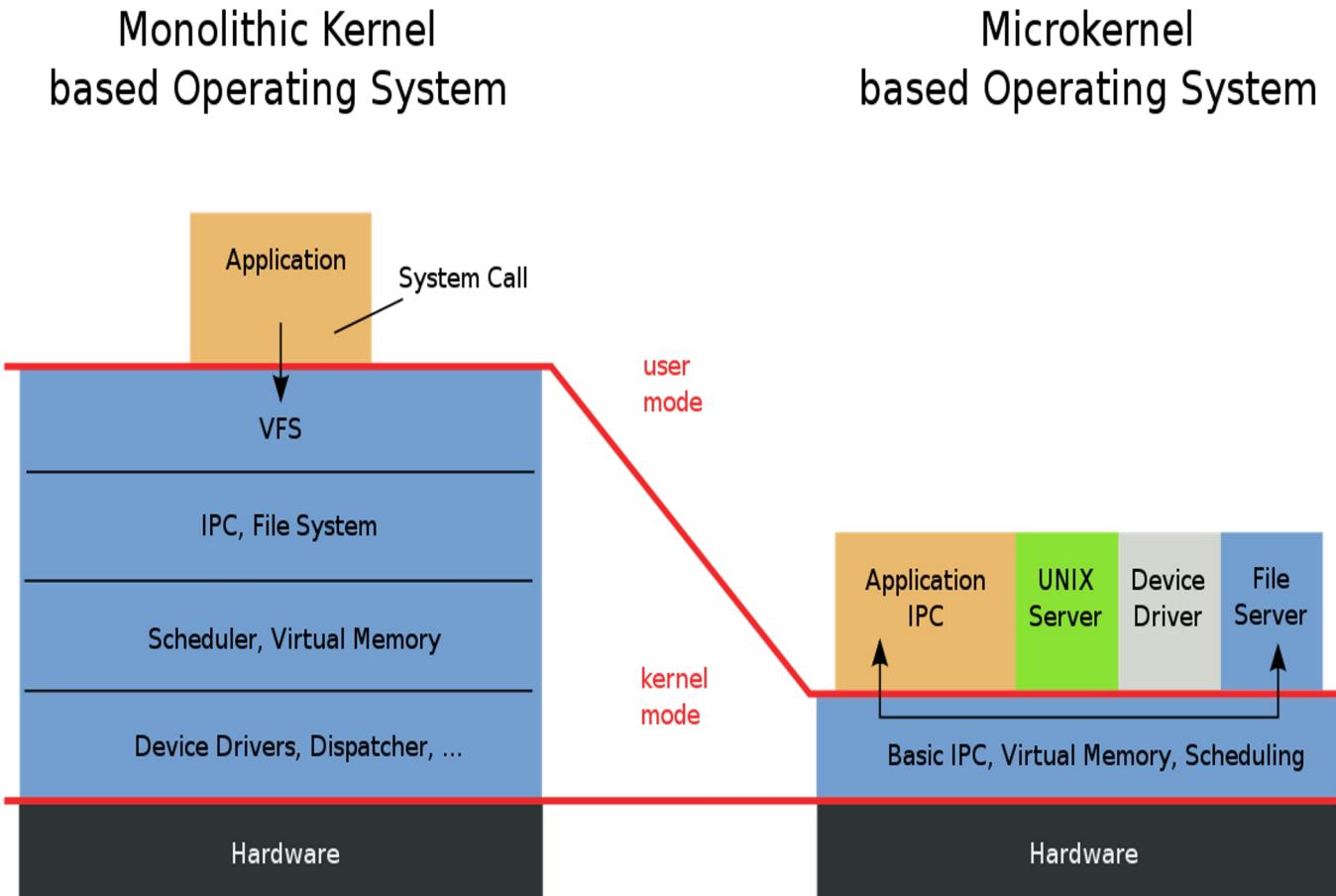
# Microkernel: Another definition

The microkernel architecture pattern consists of **two types of architecture components: a core system and plug-in modules**. Application logic is divided between independent plug-in modules and the basic core system, providing extensibility, flexibility, and isolation of application features and custom processing logic.

The **core system of the microkernel architecture pattern traditionally contains only the minimal functionality required to make the system operational**. Many operating systems implement the microkernel architecture pattern, hence the origin of this pattern's name. From a business-application perspective, the core system is often defined as the general business logic sans custom code for special cases, special rules, or complex conditional processing.

[https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch03.html#:~:text=The%20microkernel%20architecture%20pattern%20\(sometimes,a%20typical%20third%2Dparty%20product.](https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch03.html#:~:text=The%20microkernel%20architecture%20pattern%20(sometimes,a%20typical%20third%2Dparty%20product.)

# Microkernel vs. Monolithic Kernel



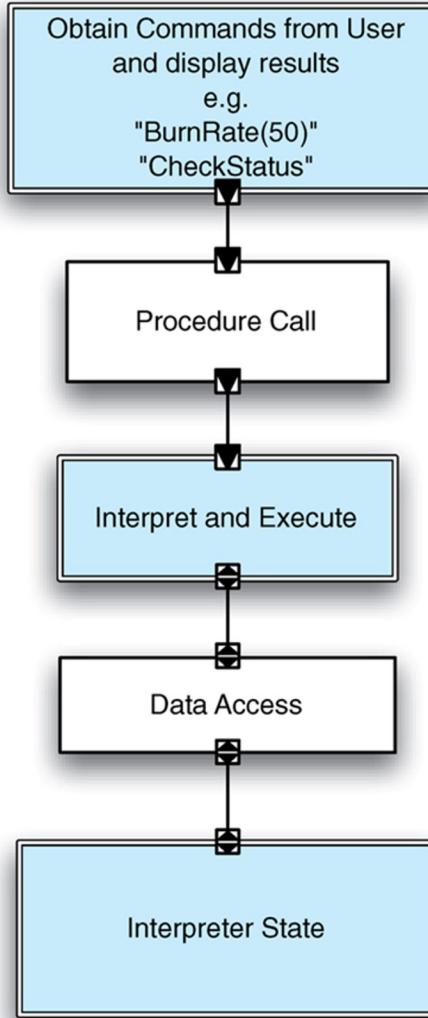
# Interpreter

A language syntax and grammar needs to be processed at runtime

An interpreter provides parsing facilities and an execution environment.

Note: there is a GoF pattern called Interpreter. It's a class structure to implement interpretation.

# Lunar Lander as an Interpreter



# Interpreter: Architecture vs. Design

What is the difference between

- The Interpreter Architecture Pattern
- The Interpreter Design Pattern

C++ Compiler: uses the Interpreter Design Pattern to build an abstract syntax tree

Python: it IS an interpreter (the Interpreter Architecture pattern)

- (of course, it also uses the Interpreter Design pattern )

# Publish-Subscribe

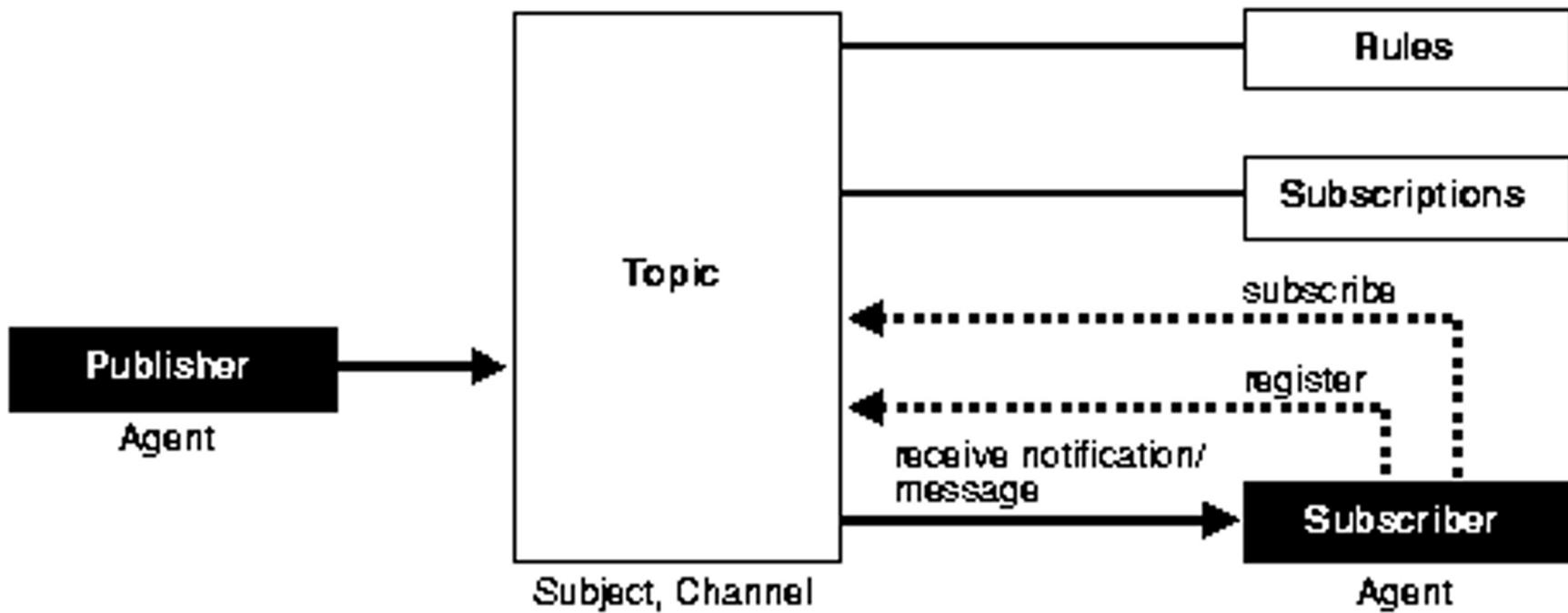
Allows event consumers (subscribers) to register for specific events

Producers publish (raise) specific events that reach a specified set of consumers

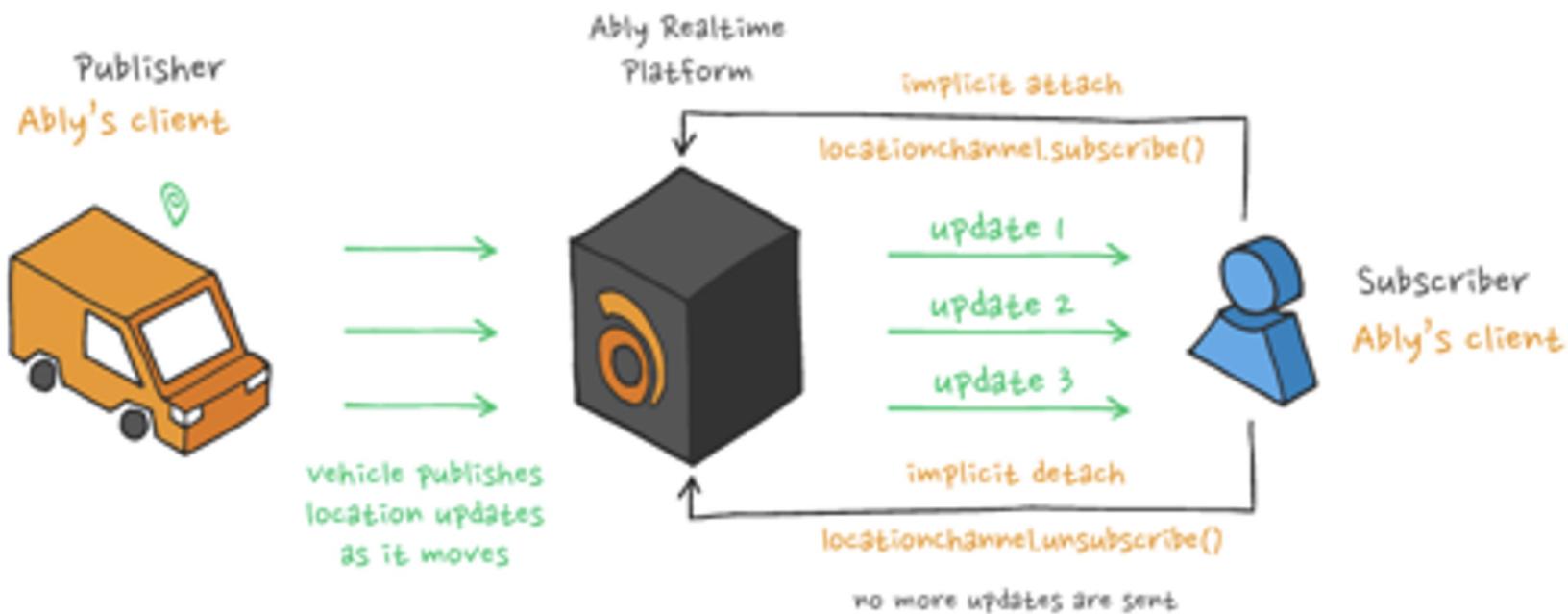
Executes a callback-operation to the event consumers

The Observer GoF pattern at the architectural level

# Publish-Subscribe



# Publish-Subscribe



# Client Server

A central server provides services to multiple clients; usually distributed

Two kinds of components: clients and servers

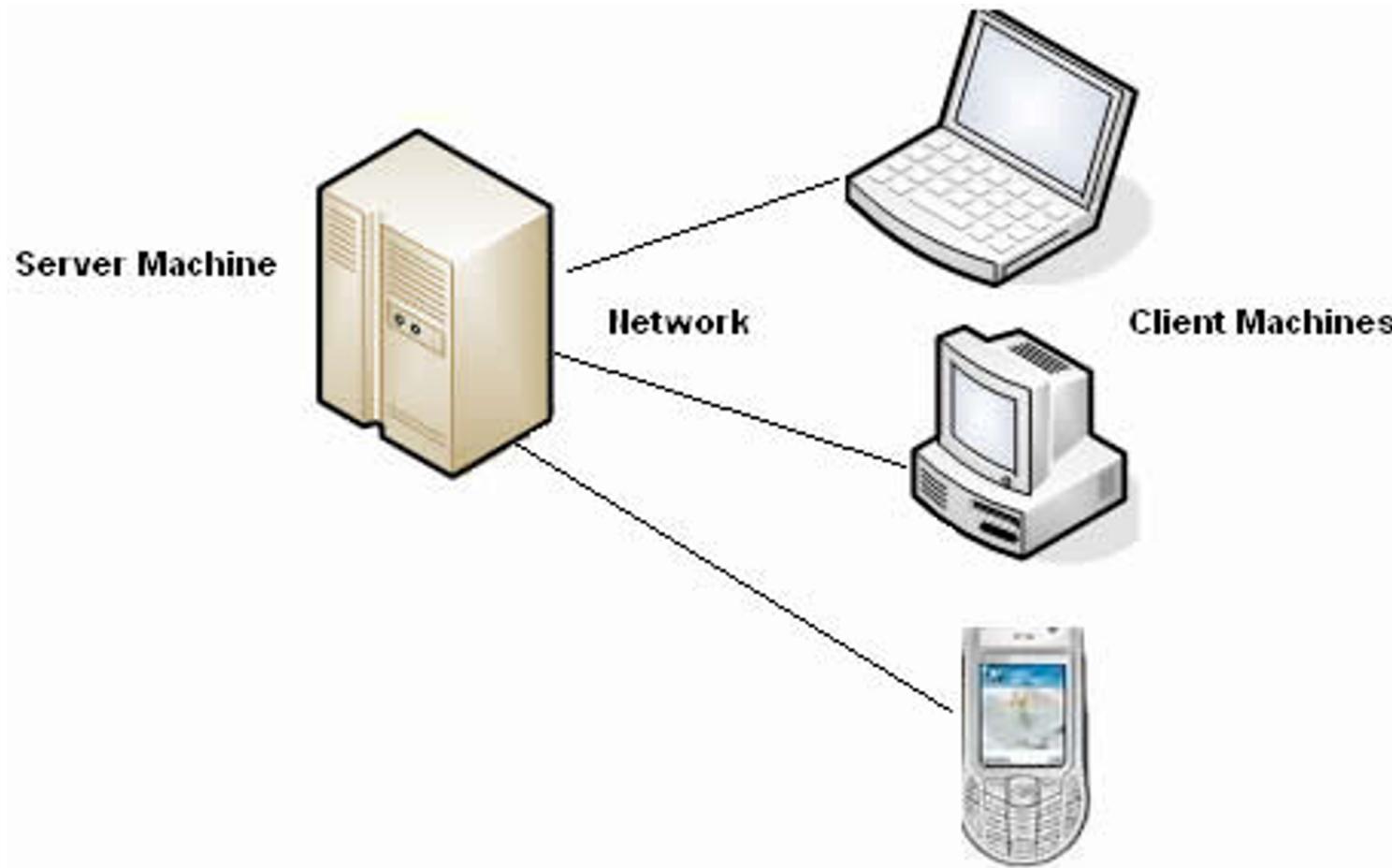
- The client requests information or services from a server.
- It needs to know an ID or address of the server.
- The server responds to requests, and processes each independently
- The server does not know the address of the client beforehand
- Clients are optimized for their application task; servers are optimized for serving multiple clients.

Extremely common, especially among distributed applications

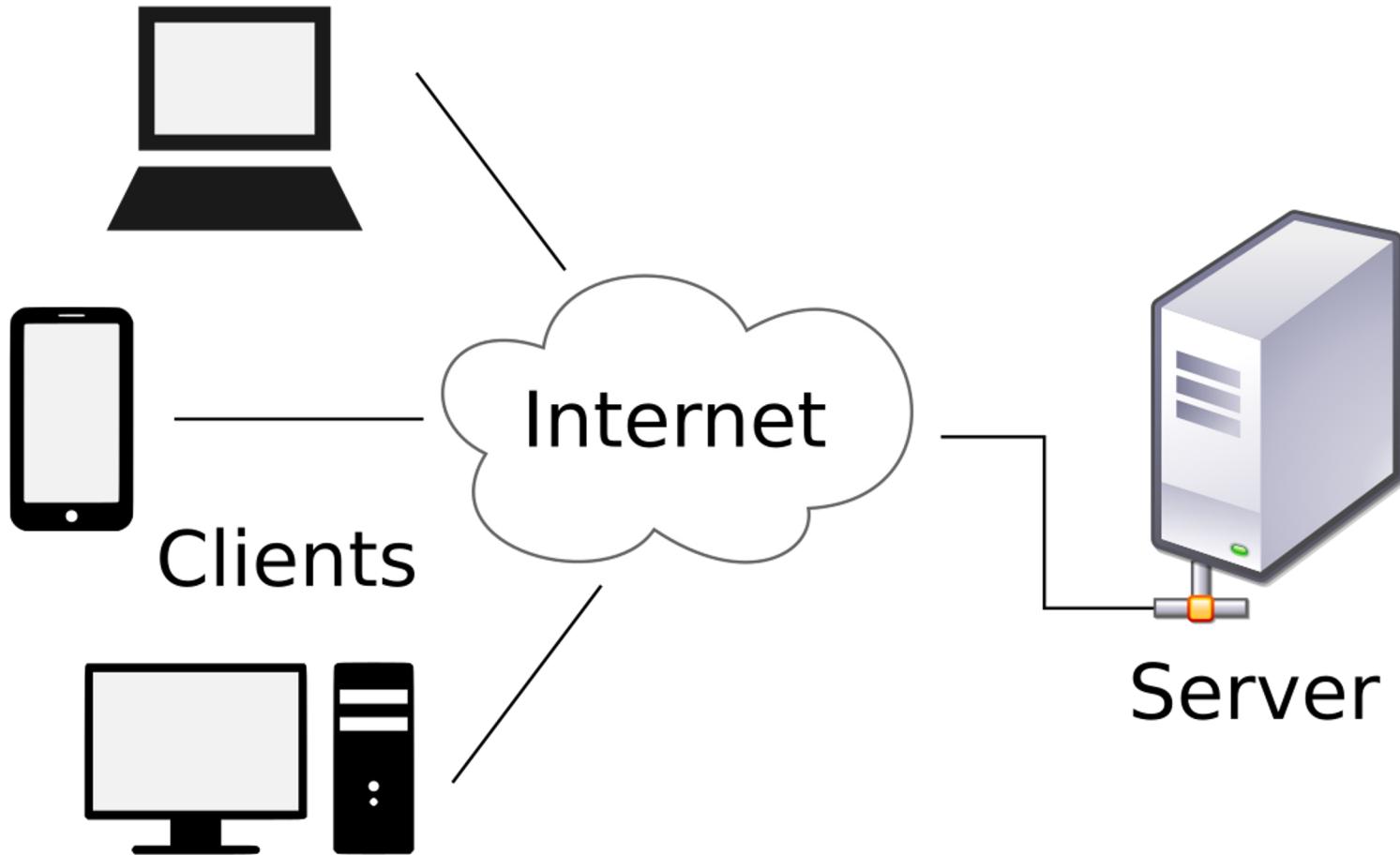
Highlights

- Usually serves a variable number of clients; may be heterogeneous
- Systems often scale well by adding more servers
- Can use multiple servers to increase availability (with some caveats)

# Client Server



# Client Server



# Peer-to-Peer

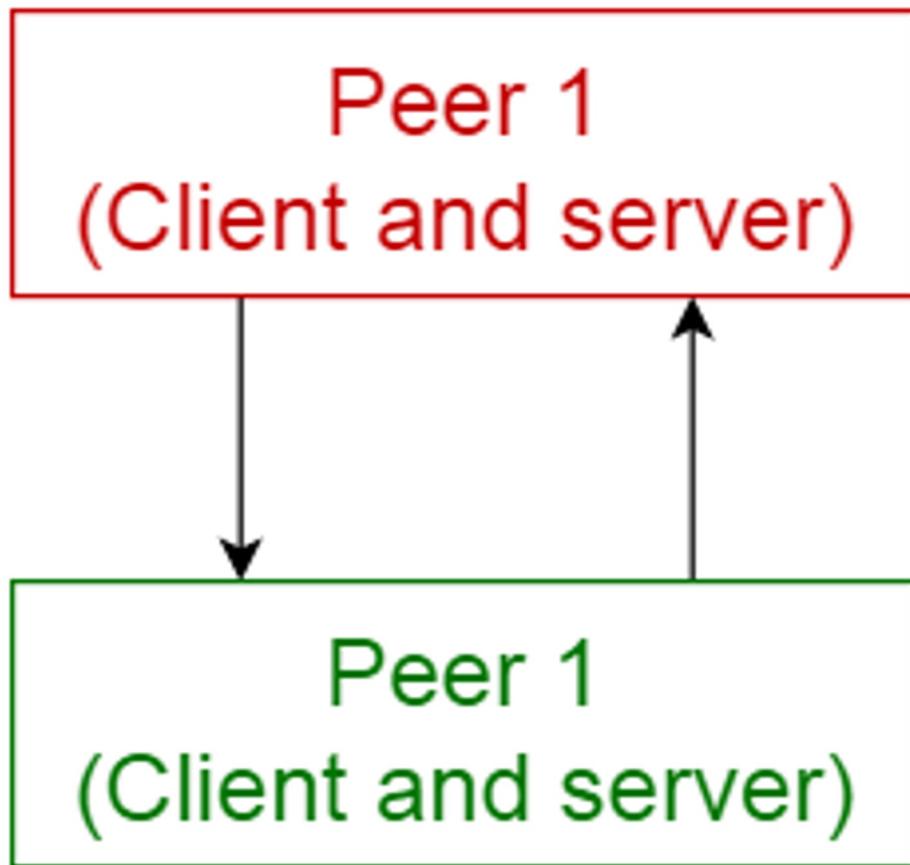
Distributed components

Each component has equal responsibilities (i.e., can act as both a client and a server)

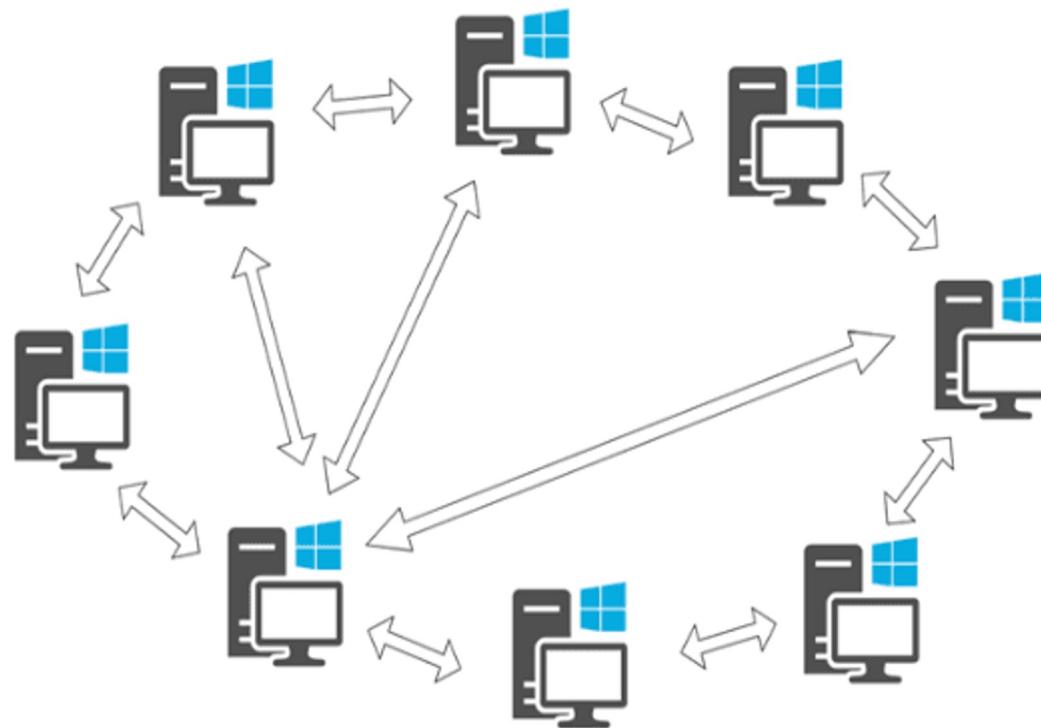
Each component offers its own services or data

A peer-to-peer network often consists of a dynamic number of components

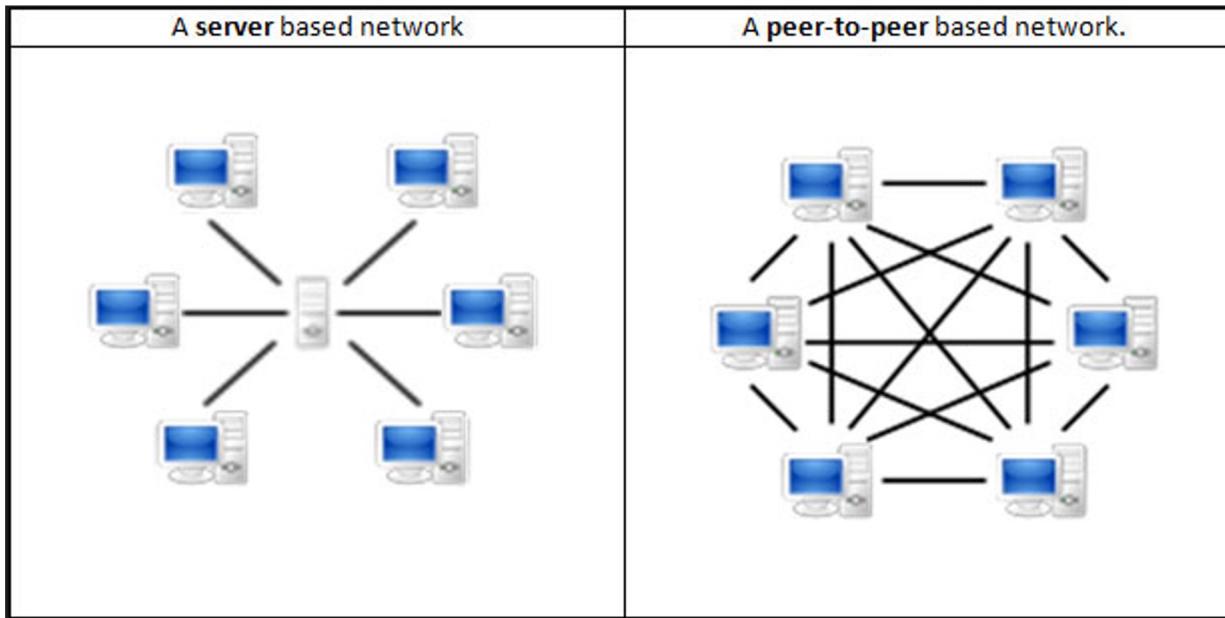
# Peer to Peer



# Peers: not every peer needs to connect with every other one



# Server and Peer-to-Peer



# **Client-Server vs. Peer-to-Peer**

Advantages and disadvantages of each?

# Peer-to-Peer

## Advantages

Easy and simple to set up only requiring a hub or a switch to connect all computers together.

You can access any file on the computer as-long as it is set to a shared folder.

If one computer fails to work all the other computers connected to it still continue to work.

## Disadvantages

Security is not good other than setting passwords for files that you don't want people to access.

If the connections are not connected to the computers properly then there can be problems accessing certain files.

It does not run efficiently if you have many computers, it is best to use two to eight computers. (But you can have overcome this in various ways!)

# Client-Server

## Advantages

A client server can be scaled up to many services that can also be used by multiple users.

Security is more advanced than a peer-to-peer network, you can have passwords to own individual profiles so that nobody can access anything when they want.

All the data is stored on the servers which generally have better security than most clients.

The server can control the access and resources better to guarantee that only those clients with the appropriate permissions may access and change data.

## Disadvantages

More expensive than a peer-to-peer network: you must pay for the start up cost.

When the server goes down or crashes all the computers connected to it become unavailable to use.

When you load the server, it starts to slow down due to the bit rate per second.

When everyone tries to do the same thing, it takes a little while for the server to do certain tasks.

# Broker

Separates the communication functionality of a distributed system from its application functionality

The Broker hides and mediates all communication between the components of a system (often clients and server)

The Broker consists of a client-side requestor to construct and forward invocations, and a server-side invoker to invoke the operations on the target object.

Most Brokers are extensions of Client-Server systems

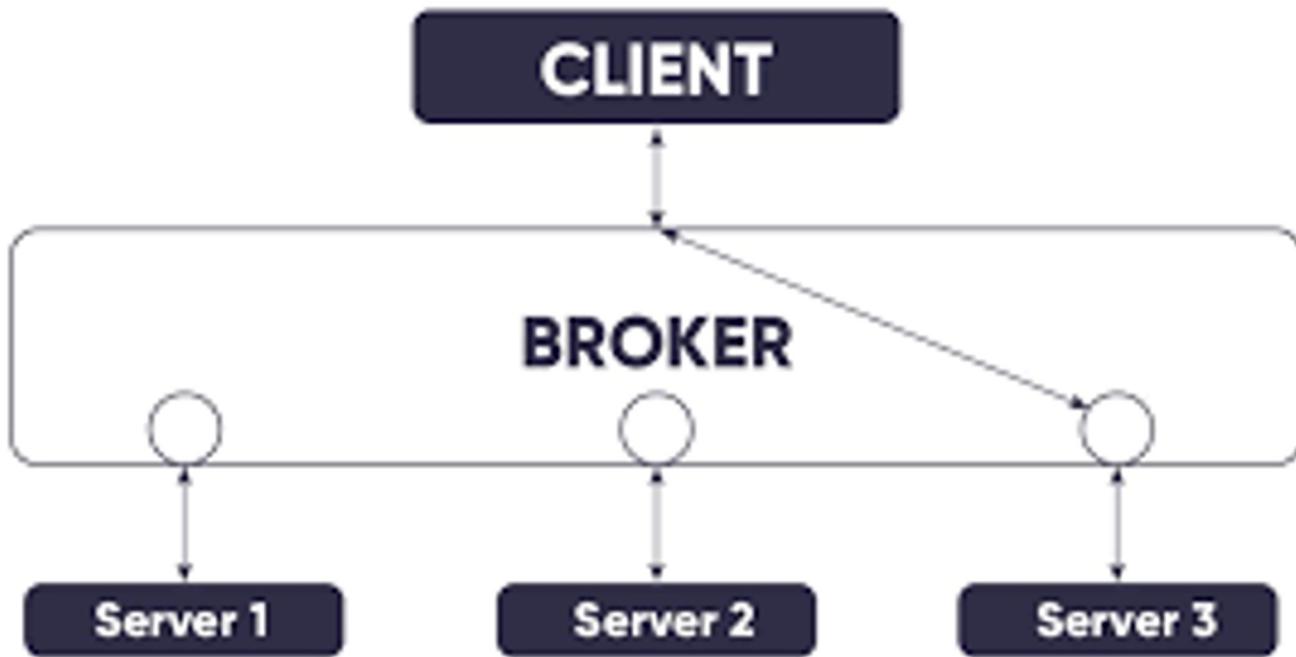
Highlights:

- Distributes them to servers for optimal performance
- Provides security, e.g., acts as a firewall or an authenticator
- Makes duplicating servers for load or availability easy

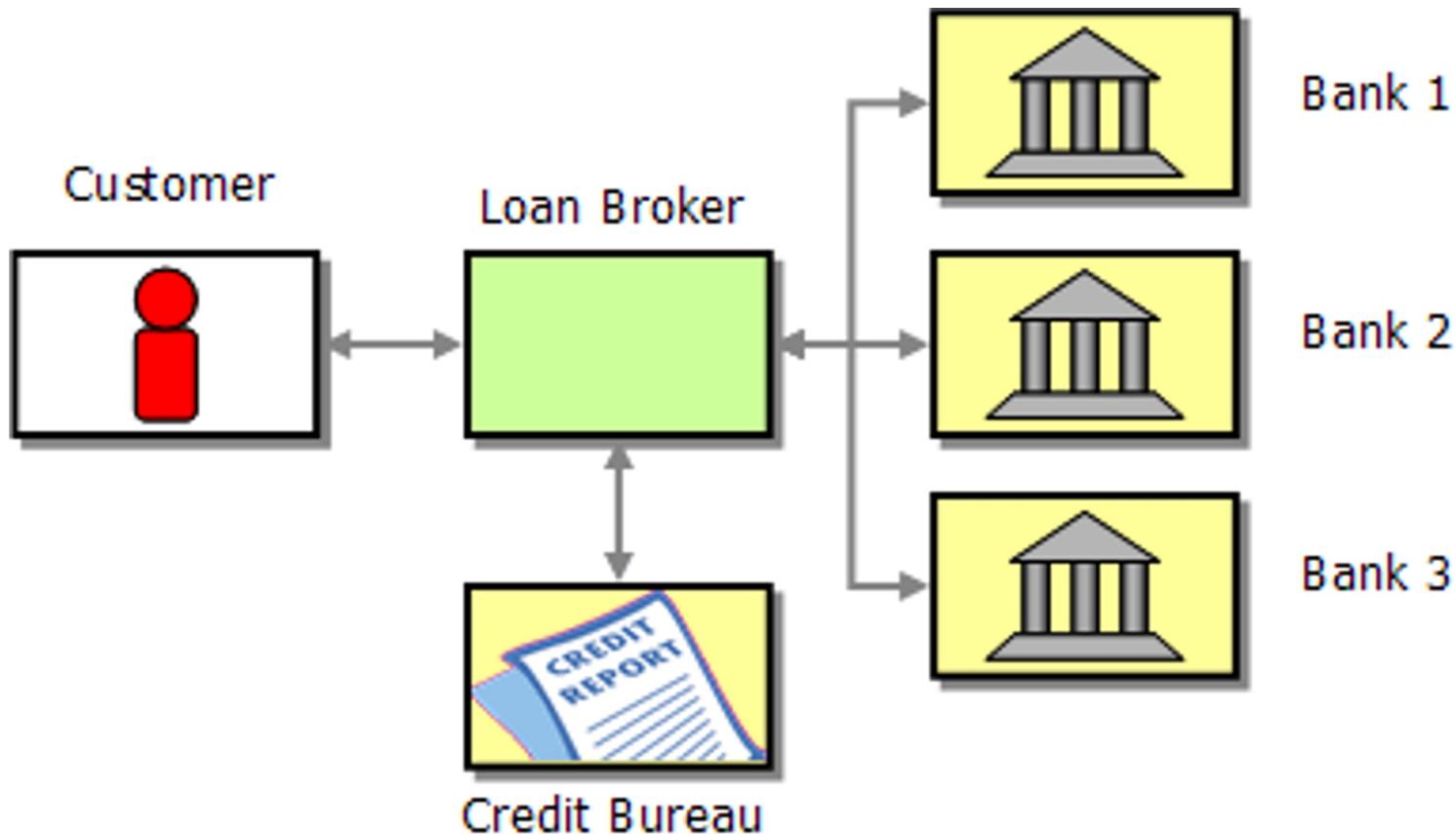
But

- Can the broker component be a bottleneck or a single point of failure?

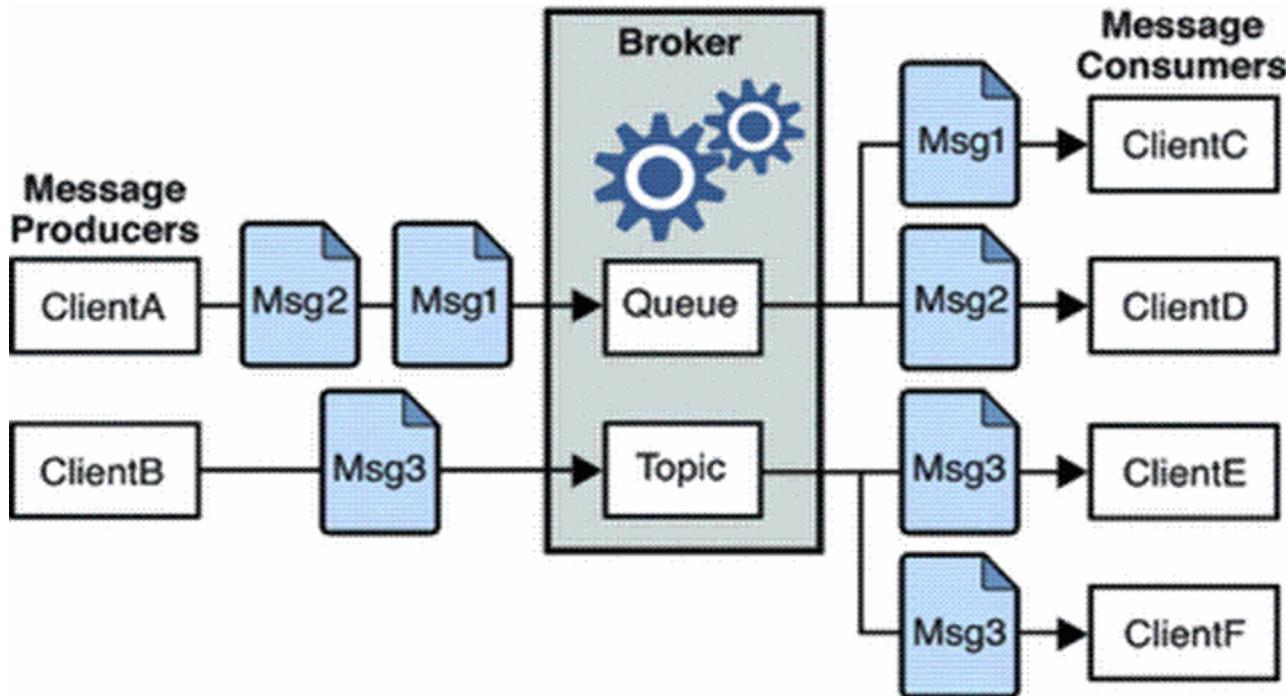
# Broker



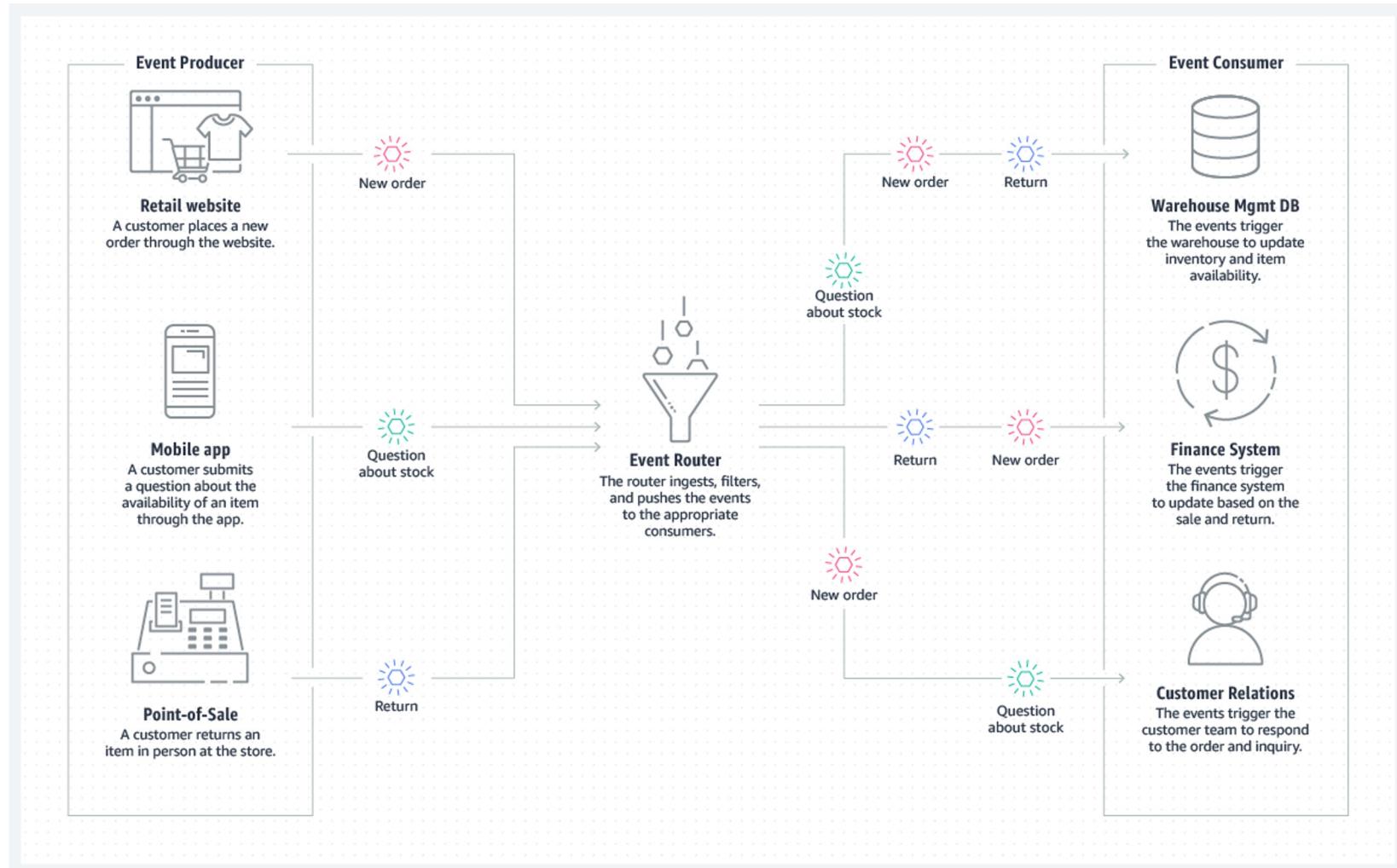
# (From Enterprise Integration Patterns)



# Broker



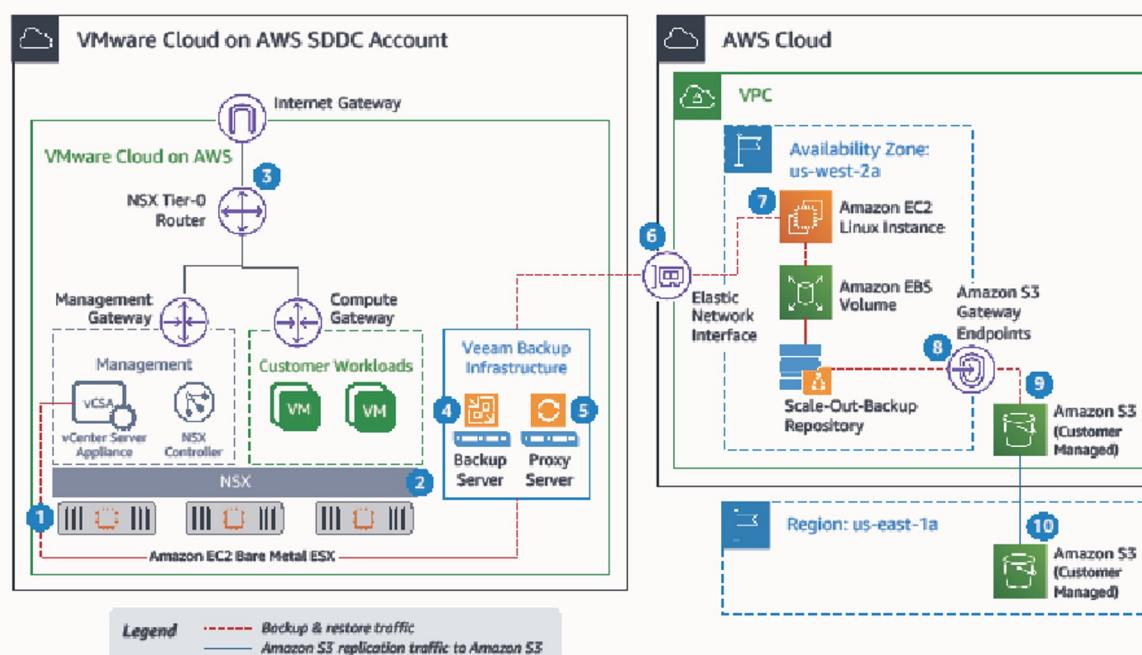
# Event driven architecture from AWS, It's also a Broker



# Broker, Complex example

## Veeam Backup on VMware Cloud on AWS

Repository options for storing backup data with Veeam using native AWS Cloud storage (Amazon EC2, Amazon EBS & Amazon S3) on the VMware Cloud platform.



- 1 Bare metal Amazon Elastic Compute Cloud (Amazon EC2) Instances running vSphere ESXi provide compute and VSAN flash storage for the workloads running on VMware Cloud on AWS.
- 2 NSX is the overlay network for VMware Cloud on AWS. It provides compute and management connectivity for workloads that run in the configured platform.
- 3 NSX Tier-0 router sends traffic from the compute & management gateways through the Internet gateway for external connectivity.
- 4 Veeam backup server is deployed as a VM in the VMC cluster. It manages backups, backup job scheduling, resource allocation, recovery verification, restore tasks, and the backup infrastructure.
- 5 Veeam proxy server processes backup and restore jobs and delivers the backup traffic through the ENI to AWS storage repositories.
- 6 Elastic Network Interface (ENI) provides fast, low-latency connections between the SDDC and the Amazon Virtual Private Cloud (Amazon VPC). Backup traffic goes through the ENI to the backup repositories in the AWS Cloud.
- 7 Daily Veeam server backups are stored on Linux-based repository servers with Amazon Elastic Block Store (Amazon EBS) storage attachments in one Availability Zone in the VPC in the Region. Repositories are configured as scale-out-backup to allow data offload from the attached Amazon Elastic Block Storage (EBS) to object storage (S3) to optimize costs.
- 8 Amazon Simple Storage Service (Amazon S3) gateway endpoints provide private access to the storage gateway service and S3 buckets.
- 9 Send backups to customer-managed S3 buckets. Configure by setting policies on the scale-out-backup repository to move data to the S3 in the Region.
- 10 Replicate backup files offsite to an S3 bucket in another Region to store data in a different Region than your workloads (when required).



© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

AWS Reference Architecture

# Shared Repository

Common component for keeping data

Accessed by multiple other components

Most databases are shared repositories these days

Considerations:

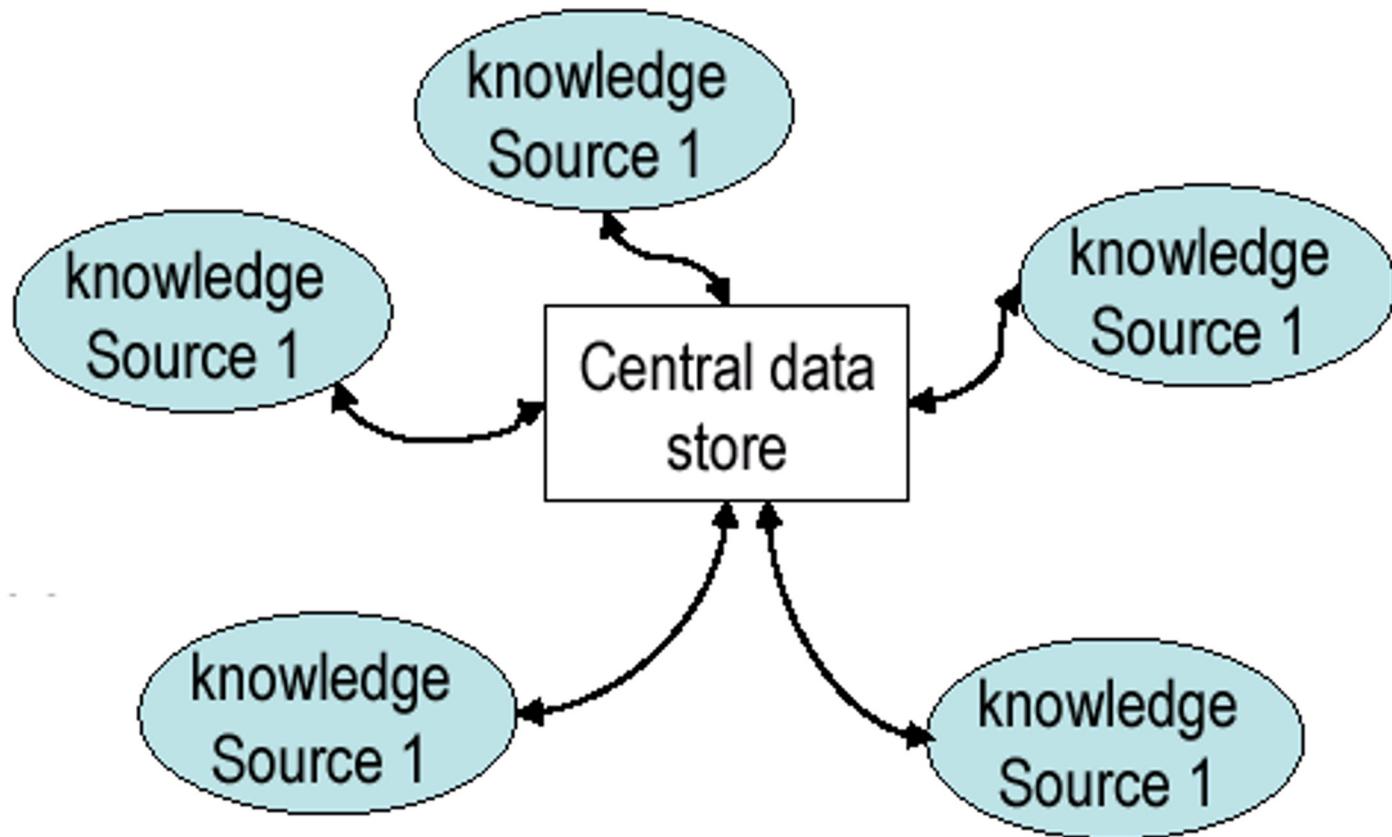
- Performance
- Concurrent access/update issues
- Reliability and availability (backups)

Variations:

- Active Repository
- Blackboard

# Shared Repository

---



# Active Repository

A repository (shared or not)

The repository does something other than just collect the data

- Maybe does some processing of the data it stores
- Maybe does some action based on the data

# Blackboard

A Blackboard is a shared repository that uses the results of its clients for heuristic computation and step-wise improvement of the solution.

Consolidates/interprets independent inputs

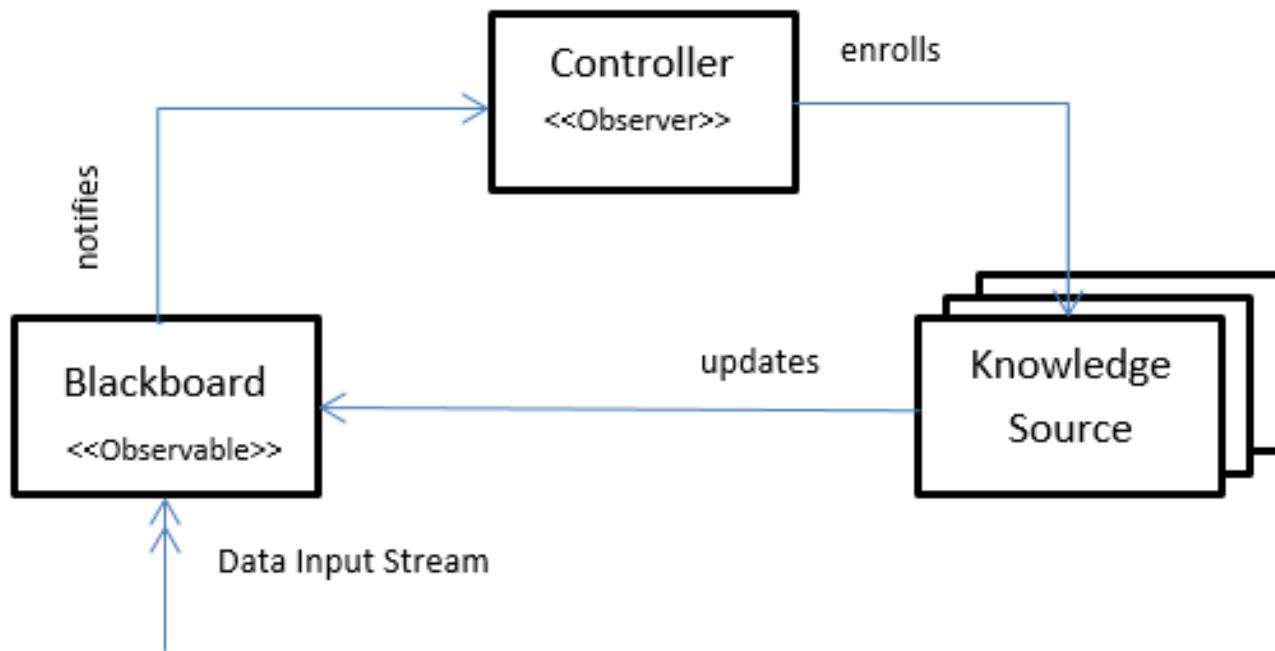
Structure:

- Shared repository with data processing capabilities
- Multiple sources of data; each can access the blackboard to check state.
- A controller/supervisor component monitors the blackboard and coordinates the clients according to the state of the blackboard.

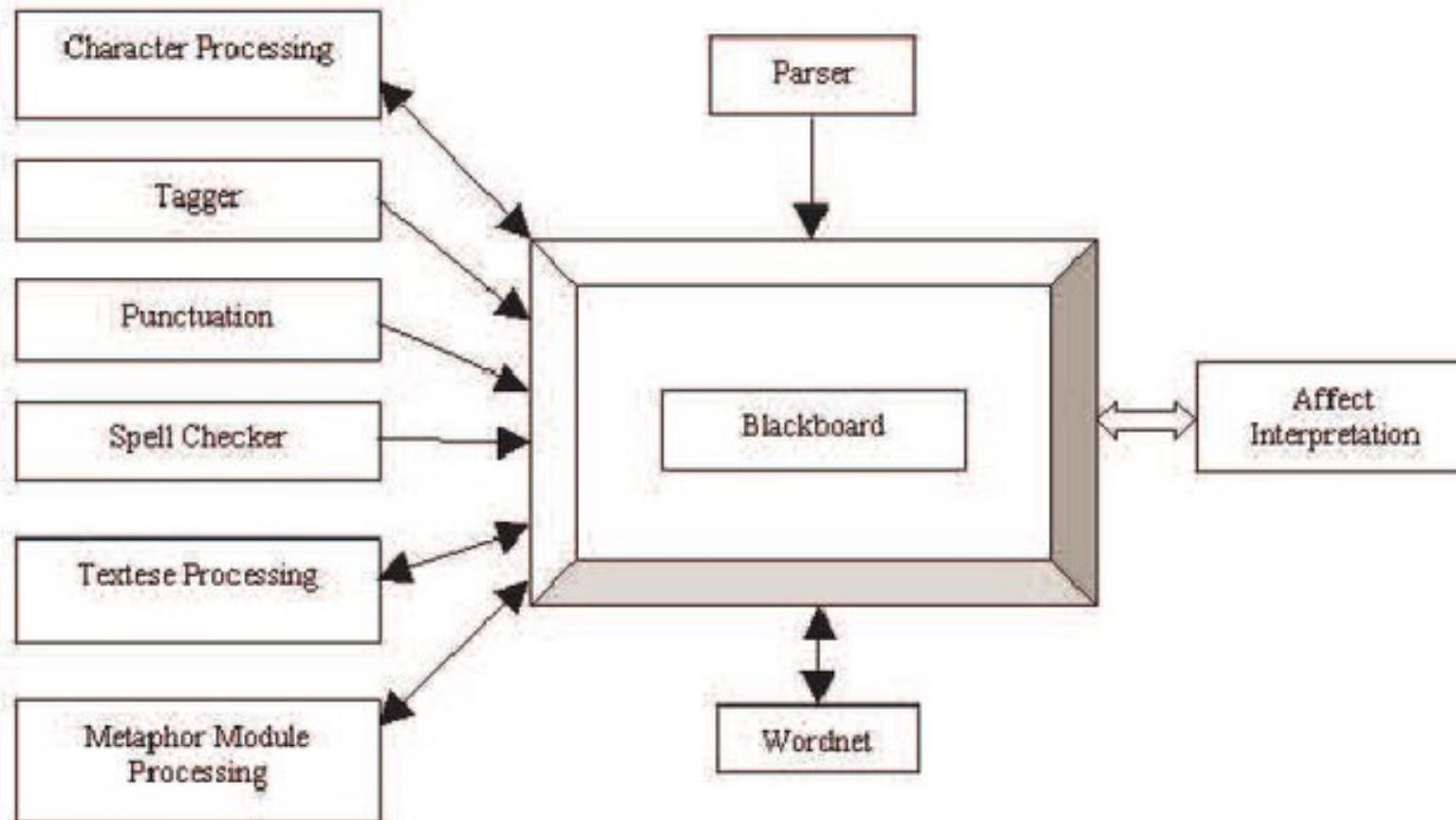
Highlights:

- Special purpose use: shared data is needed, but no deterministic solution strategies are known
- Examples: image or speech recognition
- May have challenges for debugging, enhancements

# The Blackboard Pattern



# The Blackboard Pattern



# Rule-Based System

System consists of three things: facts, rules, and an engine that acts on them.

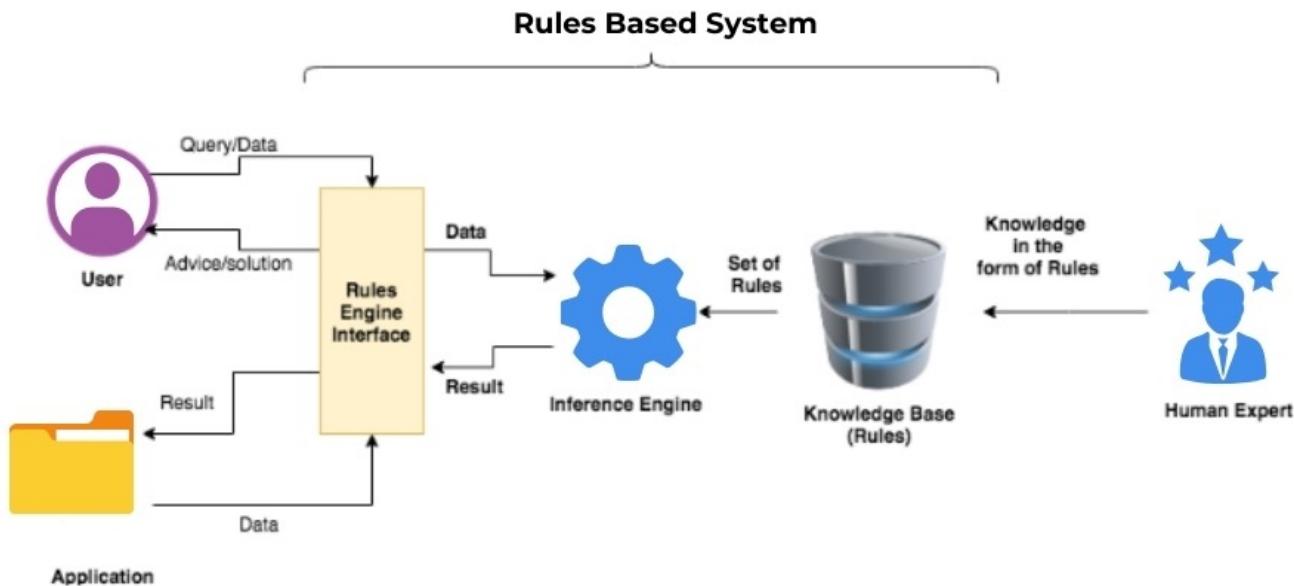
Rules are a condition and associated actions

Facts represent data

A rule-based system applies its rules to the known facts

The actions of a rule might assert new facts, which in turn trigger other rules

# Rule-Based System Example



# Model View Controller (MVC)

Managing the User Interface with 3 Components

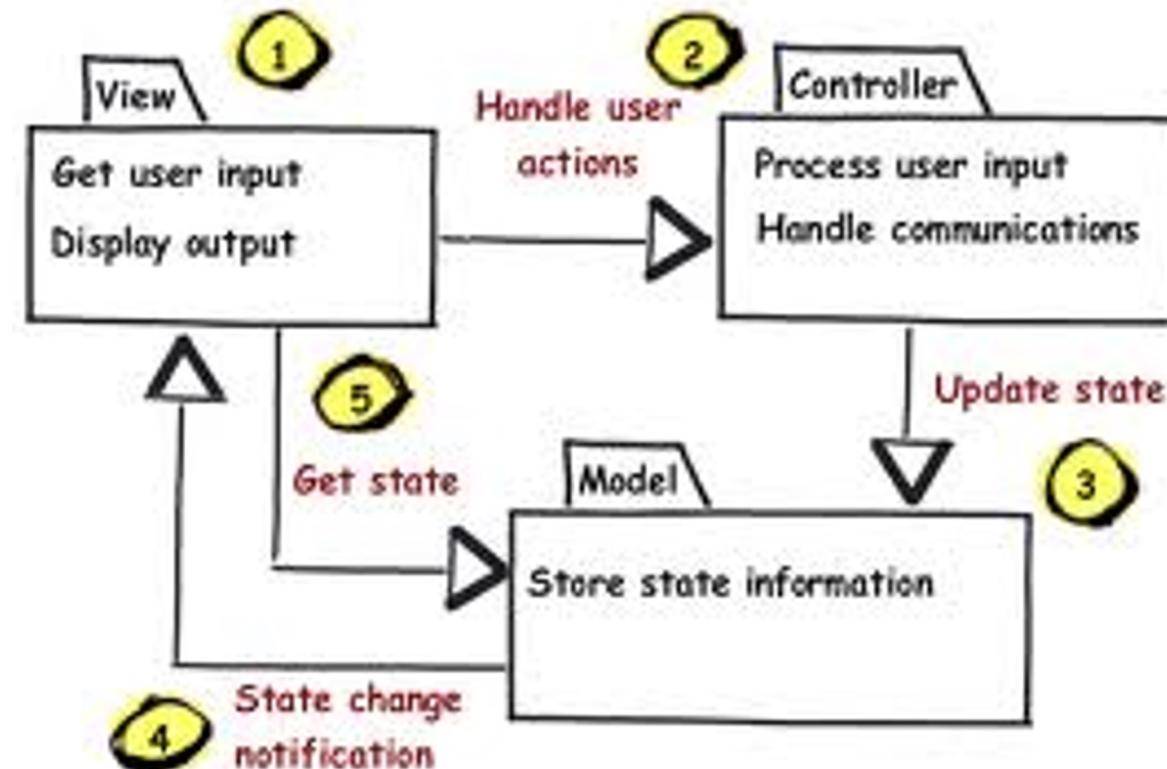
- Model: the main processing, e.g., implementation of the business rules
- View: the presentation to the user
- Controller: handles user actions and forwards them to the model

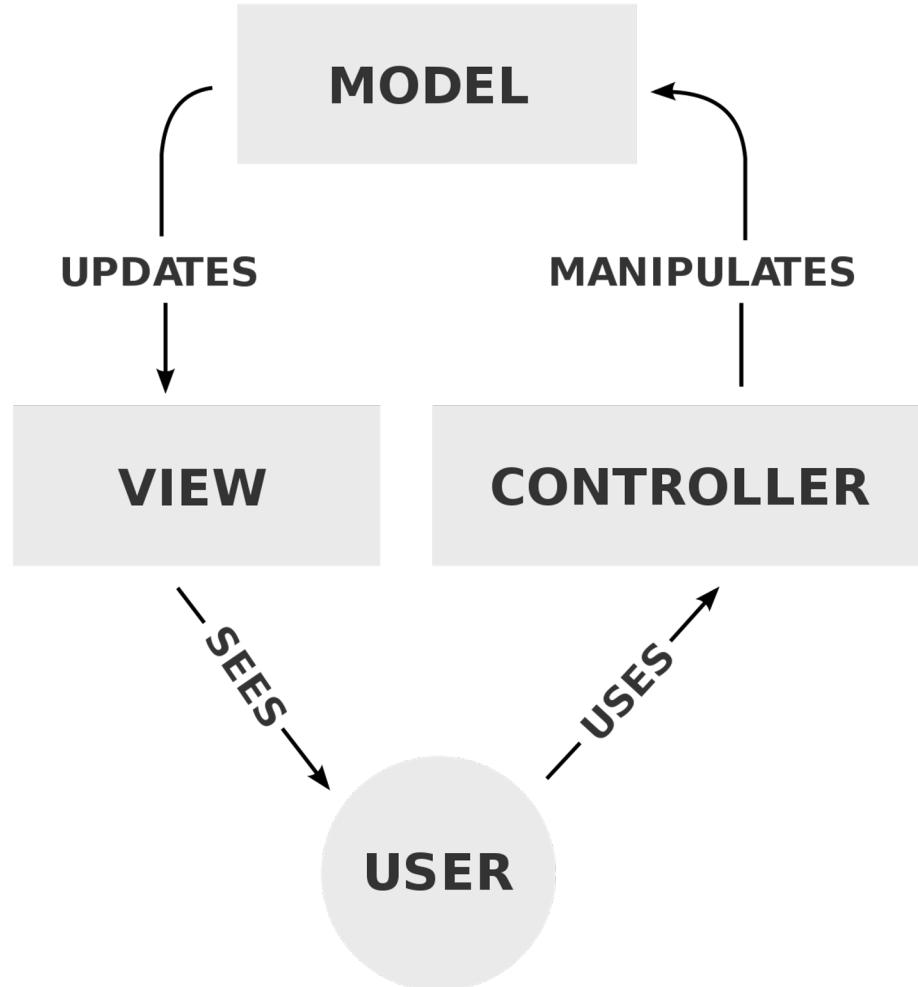
Separates the presentation from the processing

Very common these days

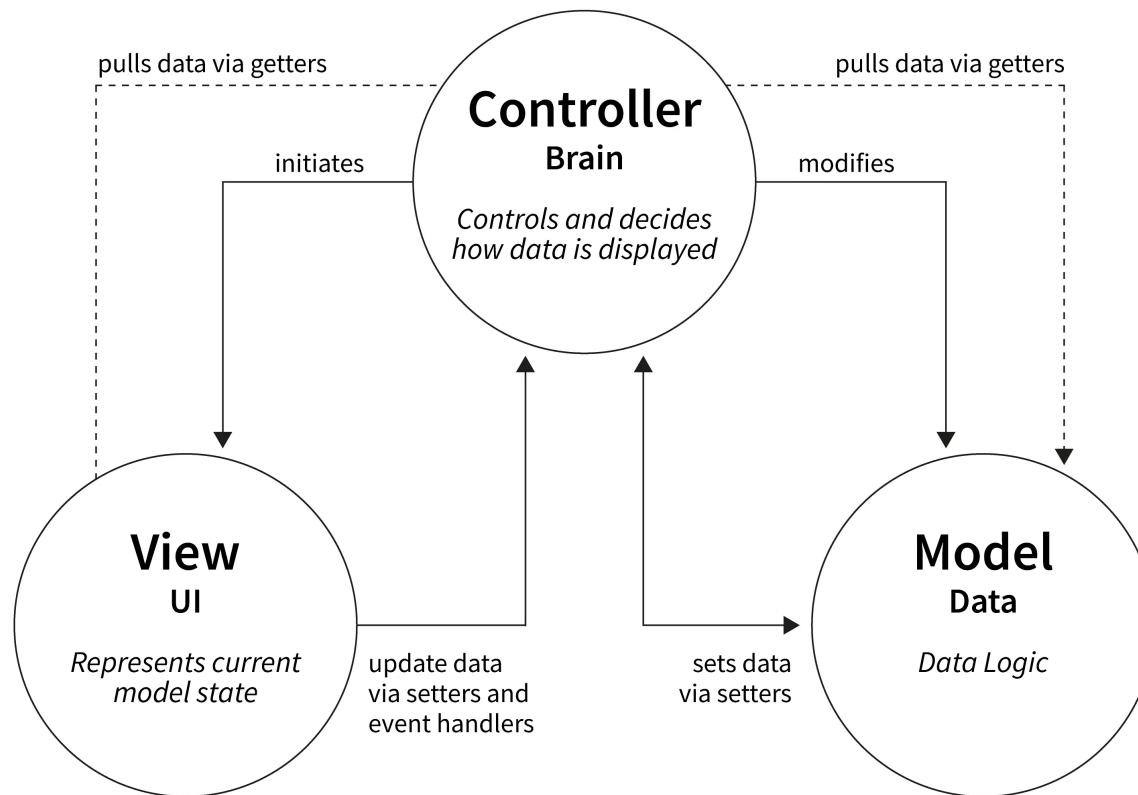
Some variants combine view and controller

# Model View Controller





# MVC Architecture Pattern



# C2 (Components and Connectors)

Multiple components that are somewhat independent and heterogeneous

A top-to-bottom hierarchy of concurrent components that interact asynchronously by sending messages through explicit connectors

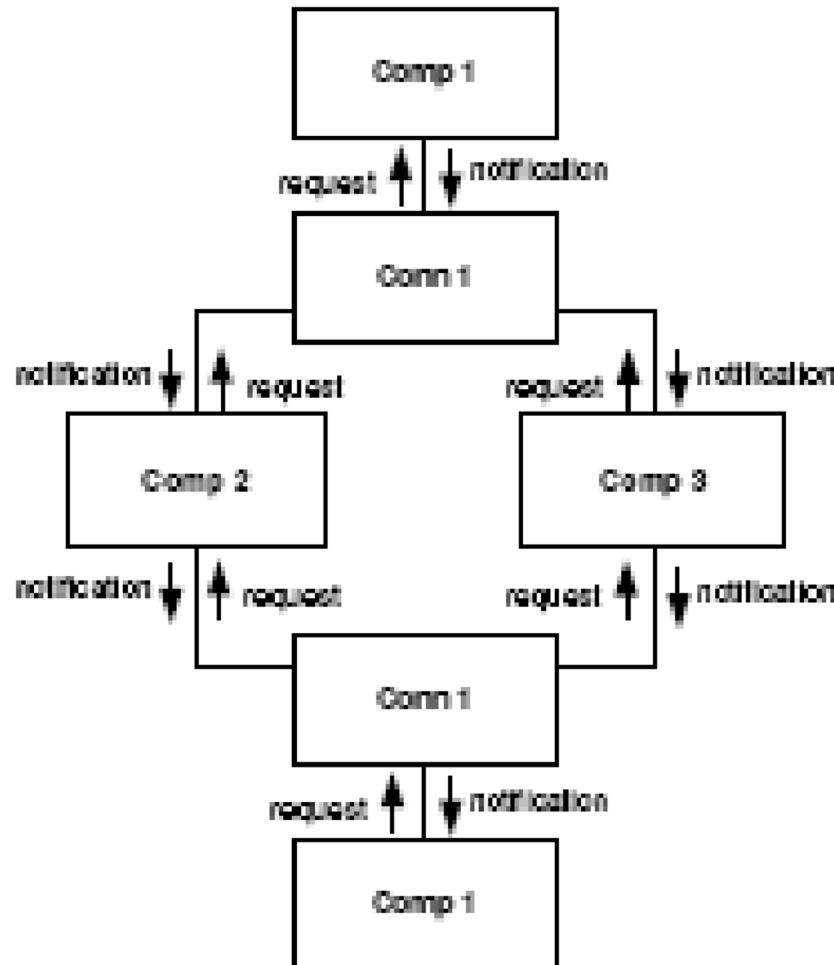
Components submit request messages upwards in the hierarchy, knowing the components above

They send notification messages down the hierarchy, without knowing the components below

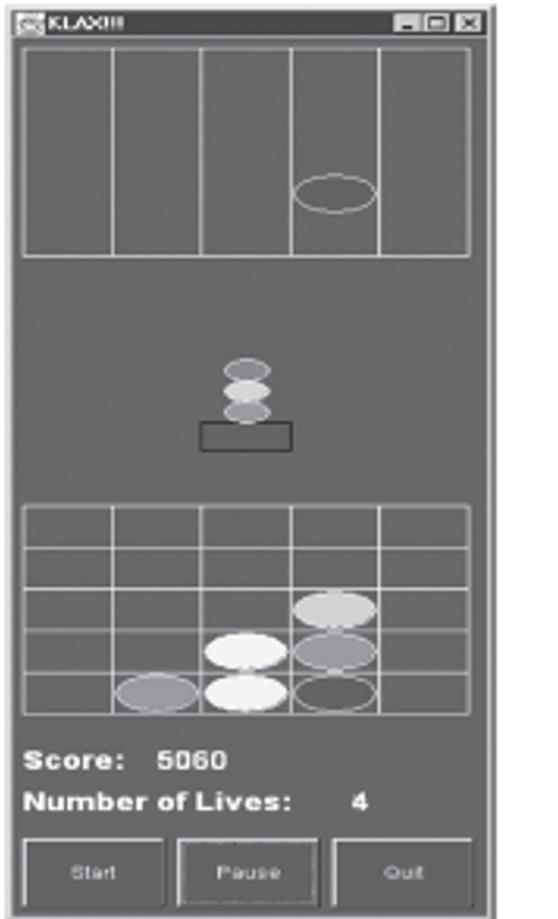
Components are connected only with connectors

The connectors broadcast, route, and filter messages

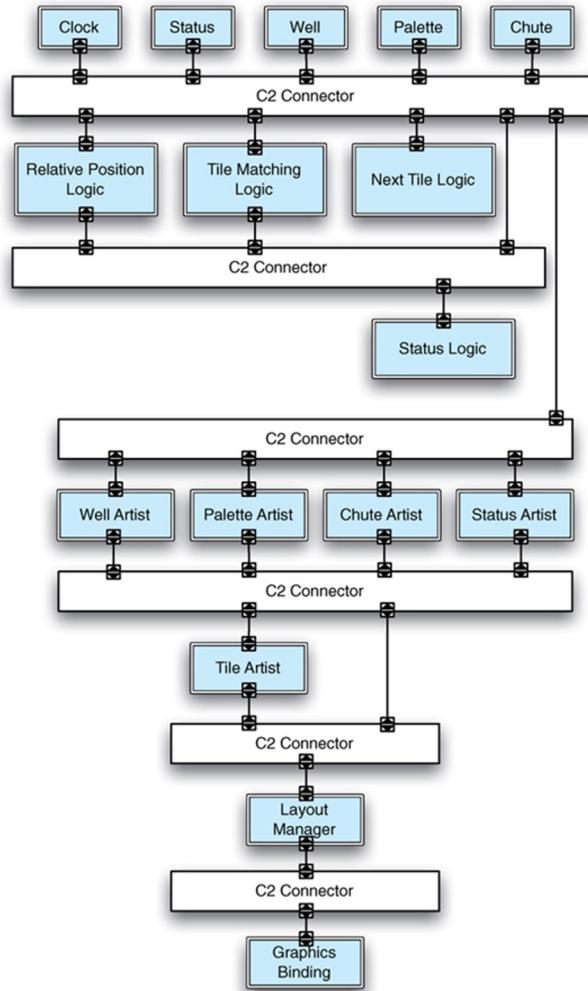
# C2



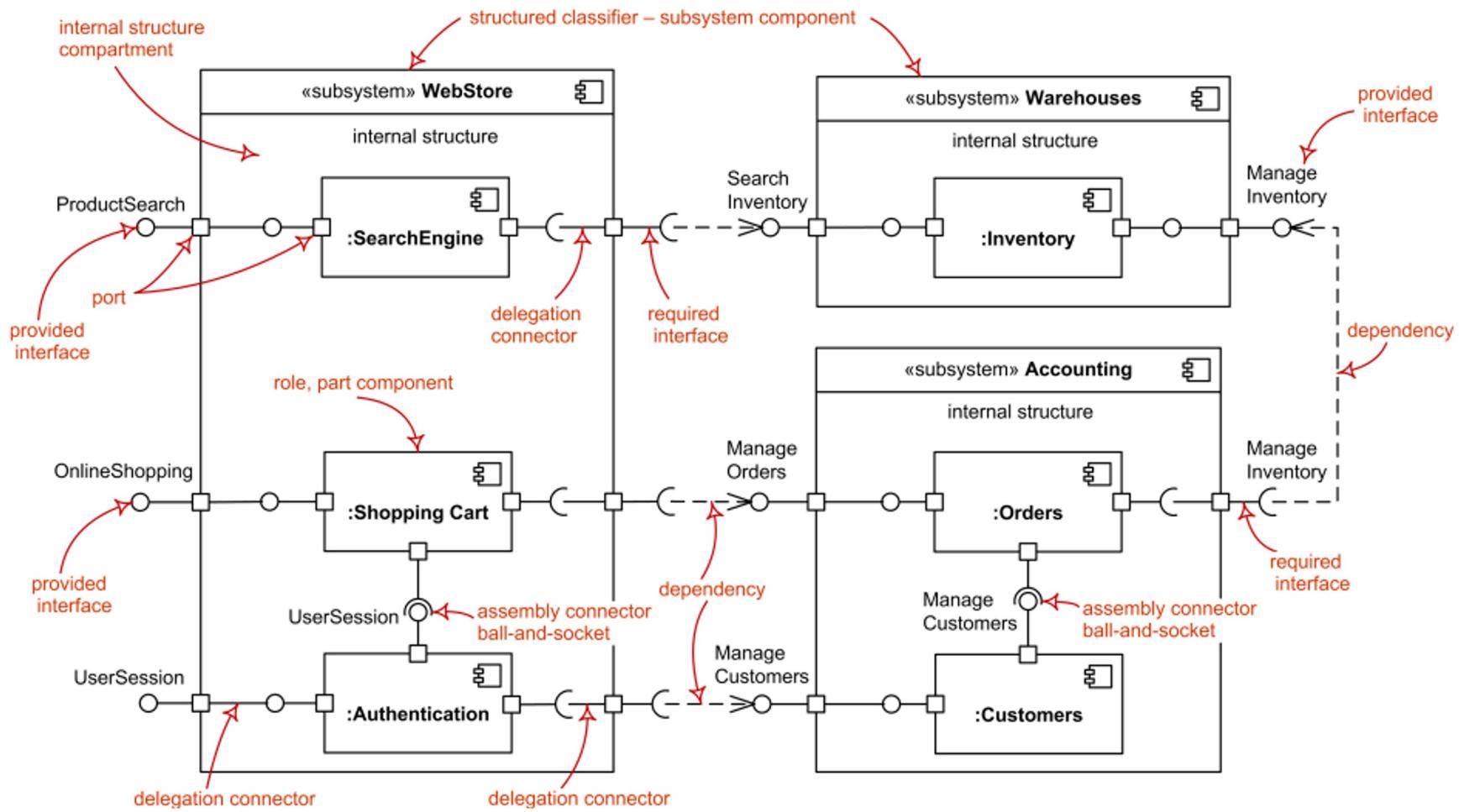
# Example: video game



# Video Game: C2



# C2



# Presentation Abstraction Control

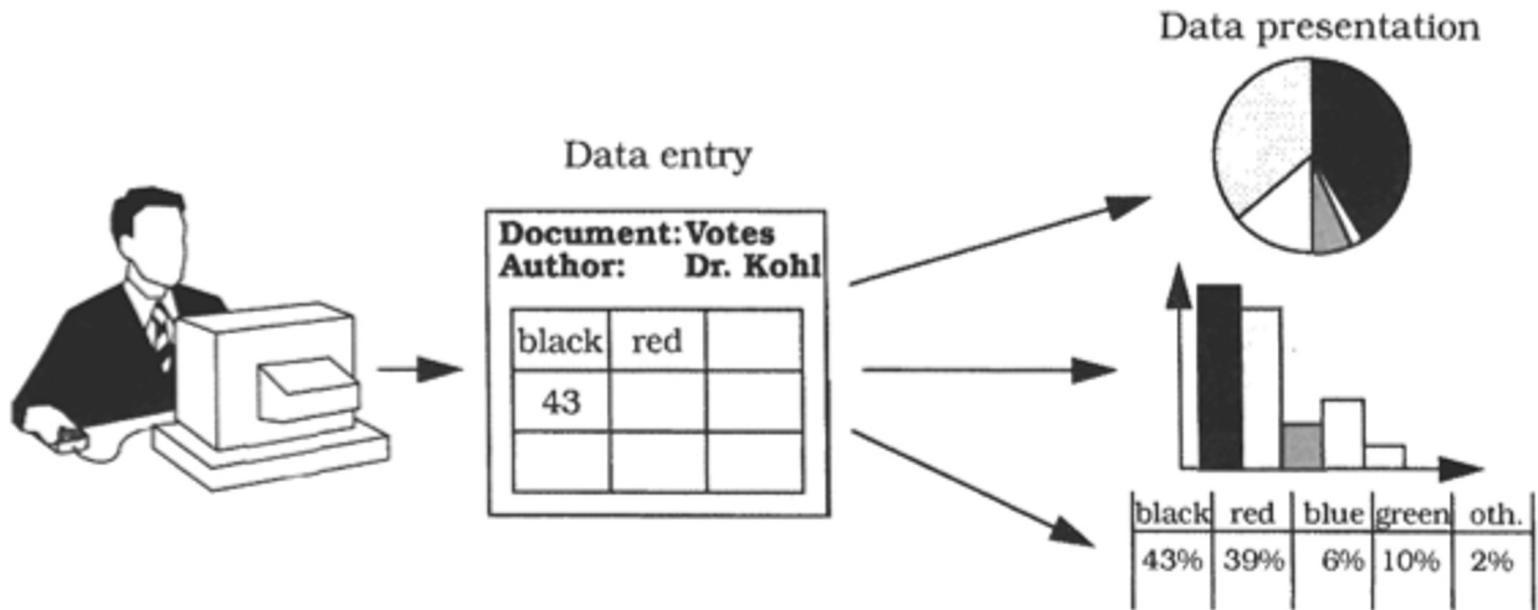
Alternative to MVC

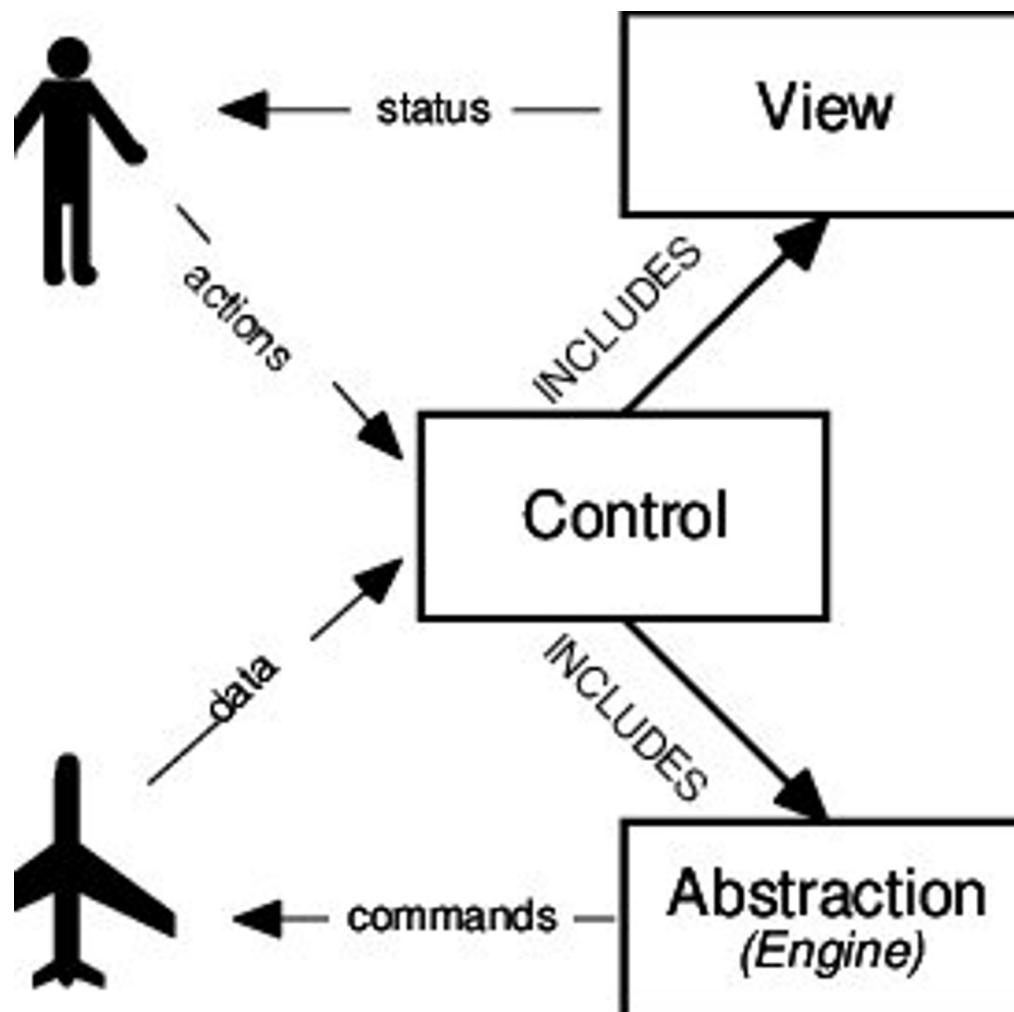
Modules with different abstractions and presentations of the data

Control: roughly equivalent to “model” and some of “control” in MVC

Handles different types of data presentations, usually simultaneously

# Presentation Abstraction Control





# Relationships of patterns

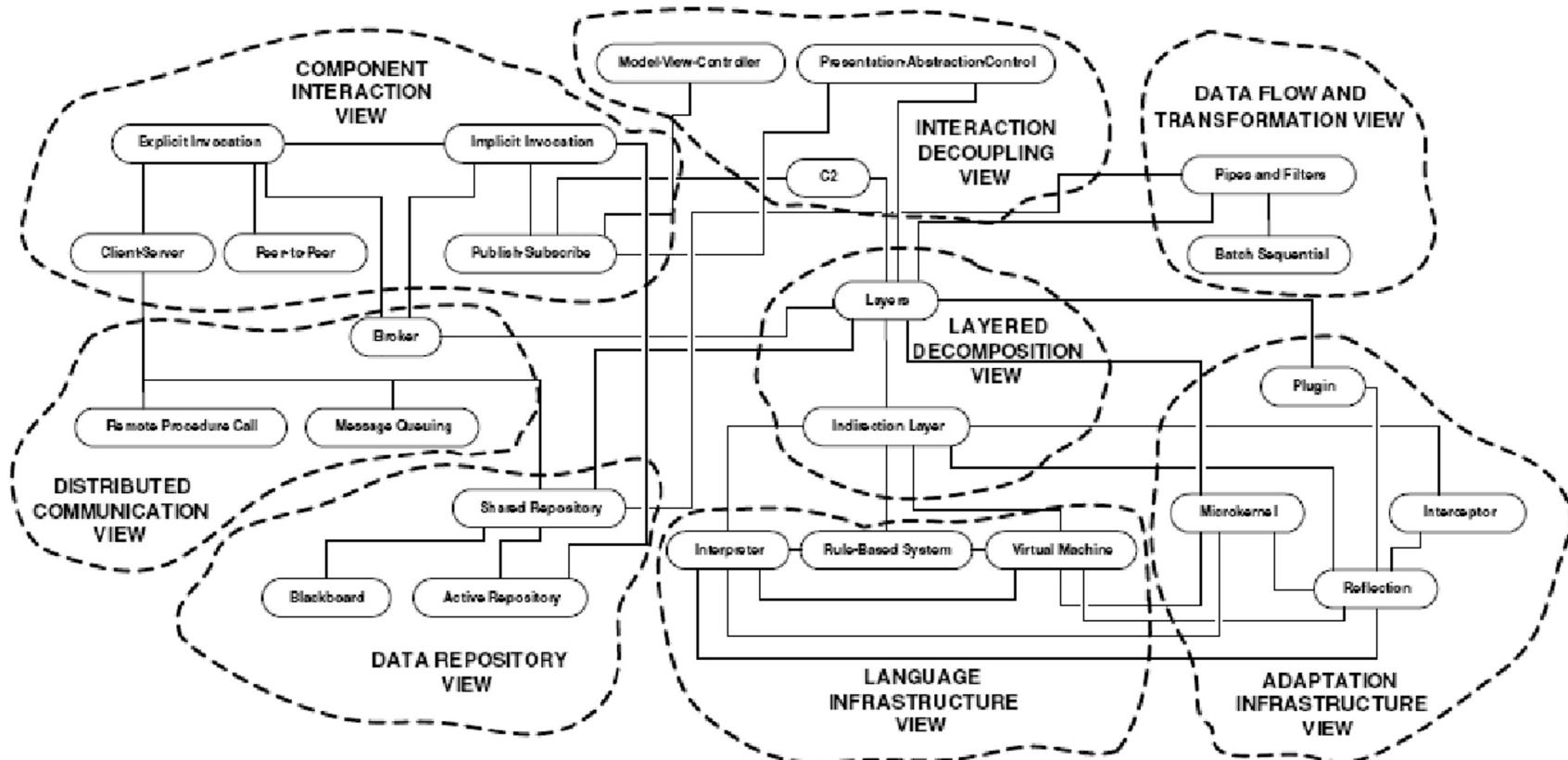


Figure 1: The patterns of the different views and the most significant pattern relationships

**Table 4-1**  
A Comparison  
Table of  
Architectural  
Styles

Style	Category &	Name	Summary	Use It When ...	Avoid It When ...
<b>Language-influenced styles</b>					
Main program and subroutines		Main	Main program controls program execution, calling multiple subroutines.	... application is small and simple.	... complex data structures needed (because of lack of encapsulation). ... future modifications likely.
Object-oriented		Object-oriented	Objects encapsulate state and accessing functions.	... close mapping between external entities and internal objects is sensible.  ... many complex and interrelated data structures.	... application is distributed in a heterogeneous network.  ... strong independence between components necessary. ... very high performance required.
<b>Layered</b>					
Virtual machines		Virtual machines	Virtual machine, or a layer, offers services to layers above it.	... many applications can be based upon a single, common layer of services.  ... interface service specification resilient when implementation of a layer must change.	... many levels are required (causes inefficiency).  ... data structures must be accessed from multiple layers.
Client-server		Client-server	Clients request service from a server.	... centralization of computation and data at a single location (the server) promotes manageability and scalability.  ... end-user processing limited to data entry and presentation.	... centrality presents a single-point-of-failure risk.  ... network bandwidth limited.  ... client machine capabilities rival or exceed the server's.
<b>Dataflow Styles</b>					
Batch-sequential		Batch-sequential	Separate programs executed sequentially, with batched input.	... problem easily formulated as a set of sequential, severable steps.	... interactivity or concurrency between components necessary or desirable.  ... random-access to data required.
Pipe-and-filter		Pipe-and-filter	Separate programs, aka filters, executed, potentially concurrently. Pipes route data streams between filters.	[as with batch-sequential] ... filters are useful in more than one application.  ... data structures easily serializable.	... interaction between components required.

**Table 4-1**  
(Continued)

Style Category & Name	Summary	Use It When ...	Avoid It When ...
<b>Shared Memory</b>			
Blackboard	Independent programs, access and communicate exclusively through a global repository known as blackboard.	... all calculation centers on a common, changing data structure. ... order of processing dynamically determined and data-driven.	... programs deal with independent parts of the common data. ... interface to common data susceptible to change. ... interactions between the independent programs require complex regulation.
Rule-based	Use facts or rules entered into the knowledge base to resolve a query.	... problem data and queries expressible as simple rules over which inference may be performed.	... number of rules is large. ... interaction between rules present. ... high-performance required.
<b>Interpreter</b>			
Interpreter	Interpreter parses and executes the input stream, updating the state maintained by the interpreter.	... highly dynamic behavior required. High degree of end-user customizability.	... high performance required.
Mobile code	Code is mobile, that is, it is executed in a remote host.	... it is more efficient to move processing to a data set than the data set to processing. ... it is desirous to dynamically customize a local processing node through inclusion of external code.	... security of mobile code cannot be assured, or sandboxed. ... tight control of versions of deployed software is required.
<b>Implicit Invocation</b>			
Publish-subscribe	Publishers broadcast messages to subscribers.	... components are very loosely coupled. ... subscription data is small and efficiently transported.	... middleware to support high-volume data is unavailable.
Event-based	Independent components asynchronously emit and receive events communicated over event buses.	... components are concurrent and independent. ... components heterogeneous and network-distributed.	... guarantees on real-time processing of events is required.

(Continued)

**Table 4-1**  
(Continued)

Style	Category &	Name	Summary	Use It When ...	Avoid It When ...
<b>Peer-to-Peer</b>					
			Peers hold state and behavior and can act as both clients and servers.	<ul style="list-style-type: none"> <li>... peers are distributed in a network, can be heterogeneous, and mutually independent.</li> <li>... robustness in face of independent failures and high scalability required.</li> </ul>	<ul style="list-style-type: none"> <li>... trustworthiness of independent peers cannot be assured or managed.</li> <li>... designated nodes to support resource discovery unavailable.</li> </ul>
<b>More Complex Styles</b>					
C2		Layered network of concurrent components communicating by events.		<ul style="list-style-type: none"> <li>... independence from substrate technologies required.</li> <li>... heterogeneous applications.</li> <li>... support for product-lines desired.</li> </ul>	<ul style="list-style-type: none"> <li>... high-performance across many layers required.</li> <li>... multiple threads are inefficient.</li> </ul>
Distributed objects		Objects instantiated on different hosts.		<ul style="list-style-type: none"> <li>... objective is to preserve illusion of location-transparency.</li> </ul>	<ul style="list-style-type: none"> <li>... high overhead of supporting middleware is excessive.</li> <li>... network properties are unmaskable, in practical terms.</li> </ul>

**table\_04\_01c**

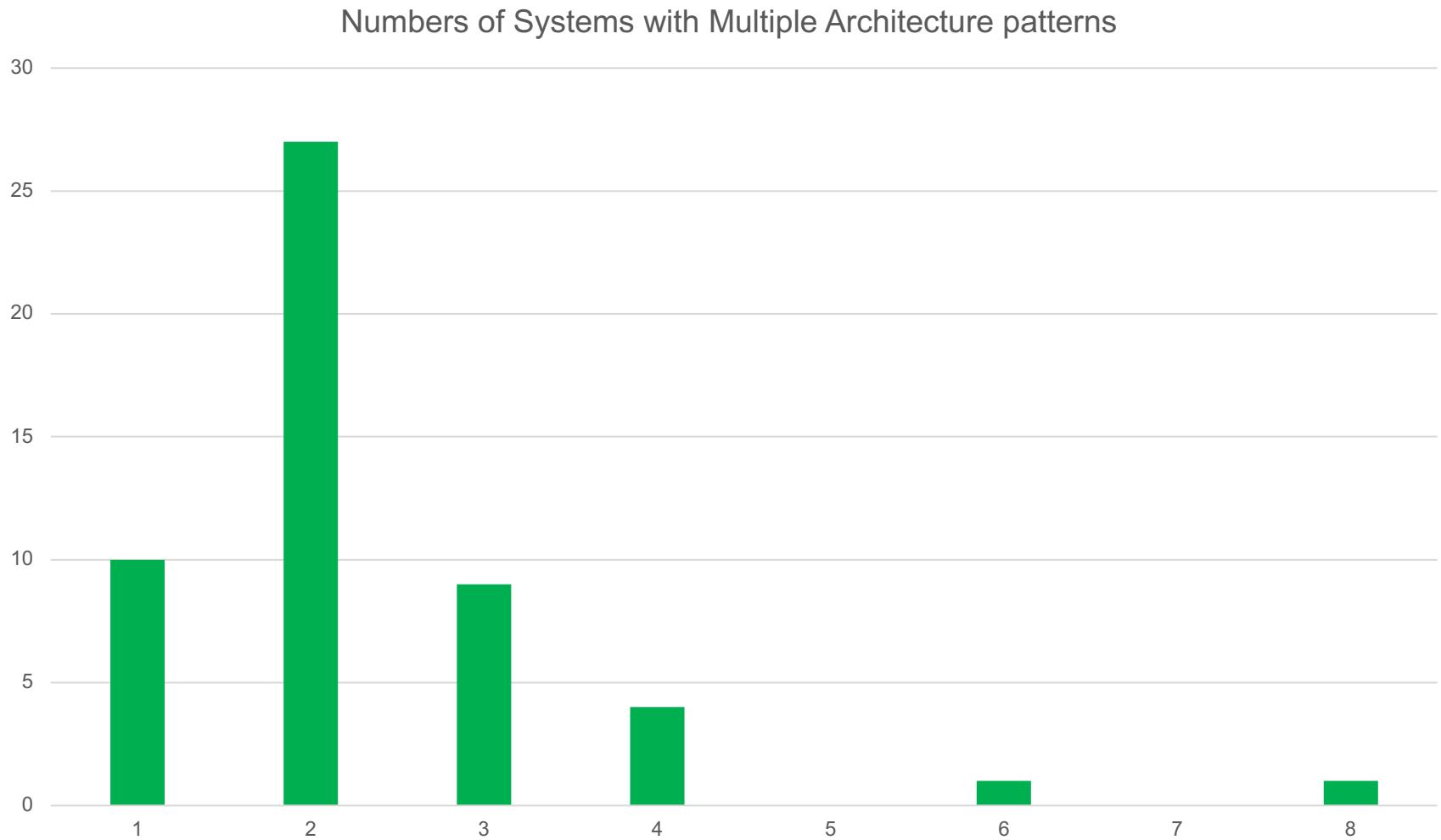
# Comparison of some patterns

Name	Advantages	Disadvantages
Layered	A lower layer can be used by different higher layers. Layers make standardization easier as we can clearly define levels. Changes can be made within the layer without affecting other layers.	Not universally applicable. Certain layers may have to be skipped in certain situations.
Client-server	Good to model a set of services where clients can request them.	Requests are typically handled in separate threads on the server. Inter-process communication causes overhead as different clients have different representations.
Master-slave	Accuracy - The execution of a service is delegated to different slaves, with different implementations.	The slaves are isolated: there is no shared state. The latency in the master-slave communication can be an issue, for instance in real-time systems. This pattern can only be applied to a problem that can be decomposed.
Pipe-filter	Exhibits concurrent processing. When input and output consist of streams, and filters start computing when they receive data. Easy to add filters. The system can be extended easily. Filters are reusable. Can build different pipelines by recombining a given set of filters	Efficiency is limited by the slowest filter process. Data-transformation overhead when moving from one filter to another.
Broker	Allows dynamic change, addition, deletion and relocation of objects, and it makes distribution transparent to the developer.	Requires standardization of service descriptions.
Peer-to-peer	Supports decentralized computing. Highly robust in the failure of any given node. Highly scalable in terms of resources and computing power.	There is no guarantee about quality of service, as nodes cooperate voluntarily. Security is difficult to be guaranteed. Performance depends on the number of nodes.
Event-bus	New publishers, subscribers and connections can be added easily. Effective for highly distributed applications.	Scalability may be a problem, as all messages travel through the same event bus
Model-view-controller	Makes it easy to have multiple views of the same model, which can be connected and disconnected at run-time.	Increases complexity. May lead to many unnecessary updates for user actions.
Blackboard	Easy to add new applications. Extending the structure of the data space is easy.	Modifying the structure of the data space is hard, as all applications are affected. May need synchronization and access control.
Interpreter	Highly dynamic behavior is possible. Good for end user programmability. Enhances flexibility, because replacing an interpreted program is easy.	Because an interpreted language is generally slower than a compiled one, performance may be an issue.

# Architecture Pattern Usage (legacy)

Pattern Name	Frequency
Layers	18
Shared Repository	13
Pipes and Filters	10
Client-Server	10
Broker	9
Model-View-Controller	7
Presentation-Abstraction- Control	7
Explicit Invocation	4
Plug-in	4
Blackboard	4
Microkernel	3
Peer to Peer	3
C2	3
Publish-Subscribe	3
State Transition	3
Interpreter	2
Half Sync, Half Async	2
Active Repository	2
Interceptor	2
Remote Procedure Call	1
Implicit Invocation	1

# Nearly every non-trivial system has multiple architecture patterns



# Frequency of pattern pairs

Pattern Pair	Frequency
Layers – Broker	6
Layers – Shared Repository	3
Pipes & Filters – Blackboard	3
Client Server – Presentation Abstraction Control	3
Layers – Presentation Abstraction Control	3
Layers – Model View Controller	3
Broker – Client-Server	2
Shared Repository – Presentation Abstraction Control	2
Layers – Microkernel	2
Shared Repository – Model View Controller	2
Client Server – Peer to Peer	2
Shared Repository – Peer to Peer	2
Shared Repository – C2	2
Peer to Peer – C2	2
Layers – Interpreter	2
Layers – Client Server	2
Pipes & Filters – Client Server	2
Pipes & Filters – Shared Repository	2
Client Server – Blackboard	2
Broker – Shared Repository	2
Broker – Half Sync/Half Async	2
Shared Repository – Half Sync/Half Async	2
Client Server – Half Sync/Half Async	2

# **Hunting Patterns in the Wild**

Patterns as they appear in architectures

# Finding Architecture Patterns

Are architecture patterns really used?

Yes!

- In a study of 45 architectures, we found architecture patterns in every architecture
- And in many cases, it is likely the architects didn't even use them on purpose

Studying architectures can help us see how to use architecture patterns

# Boxology: a Field Guide

Most architecture diagrams consist of:

- Boxes
- Lines between the boxes
- Words, usually labeling the boxes and lines

They usually follow no standard methodology

- We are generally left to figure out the meanings of the boxes and lines
- They may mean different things – in the same diagram!

But it usually works pretty well

# Exercise: Find the Patterns

You will see pictures of real architectures

Can you identify any patterns in them?

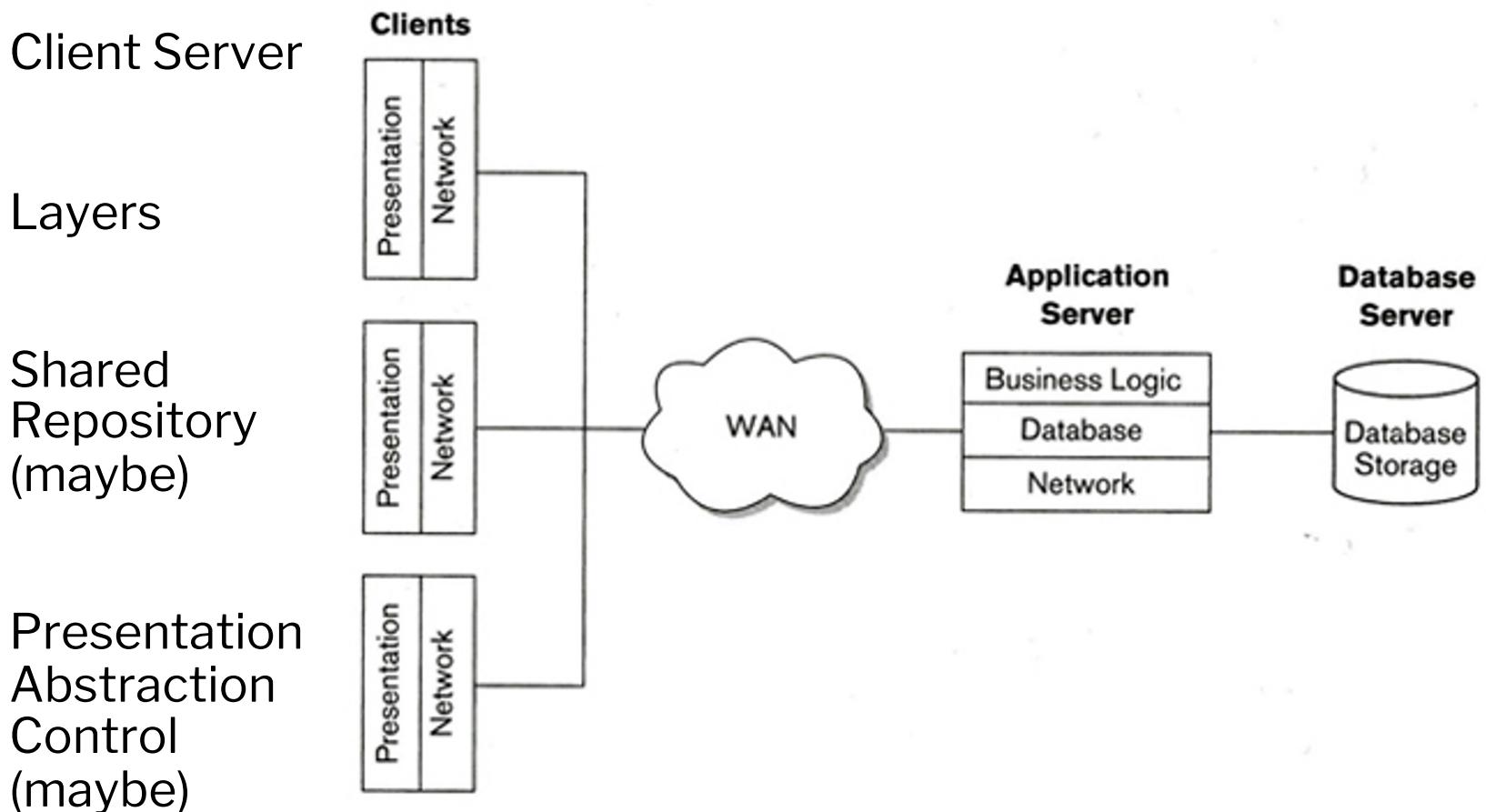
Some architectures may have more than one pattern

- (our experience: nearly all had two or more patterns)

Break into groups and discuss each

- Then we will discuss together

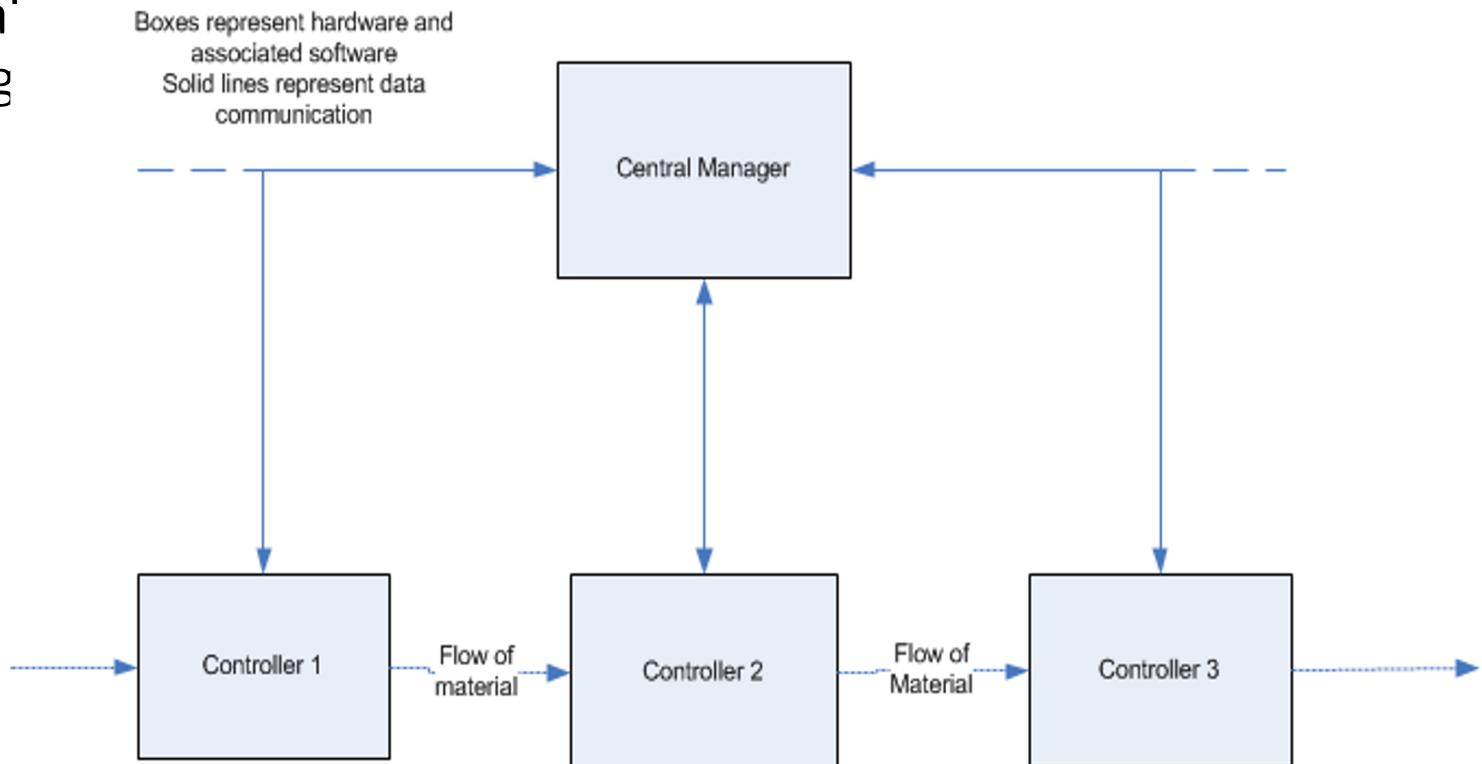
# Airline Booking System



# Materials Processing System

## Pipes and Filters

What is the  
central  
manag

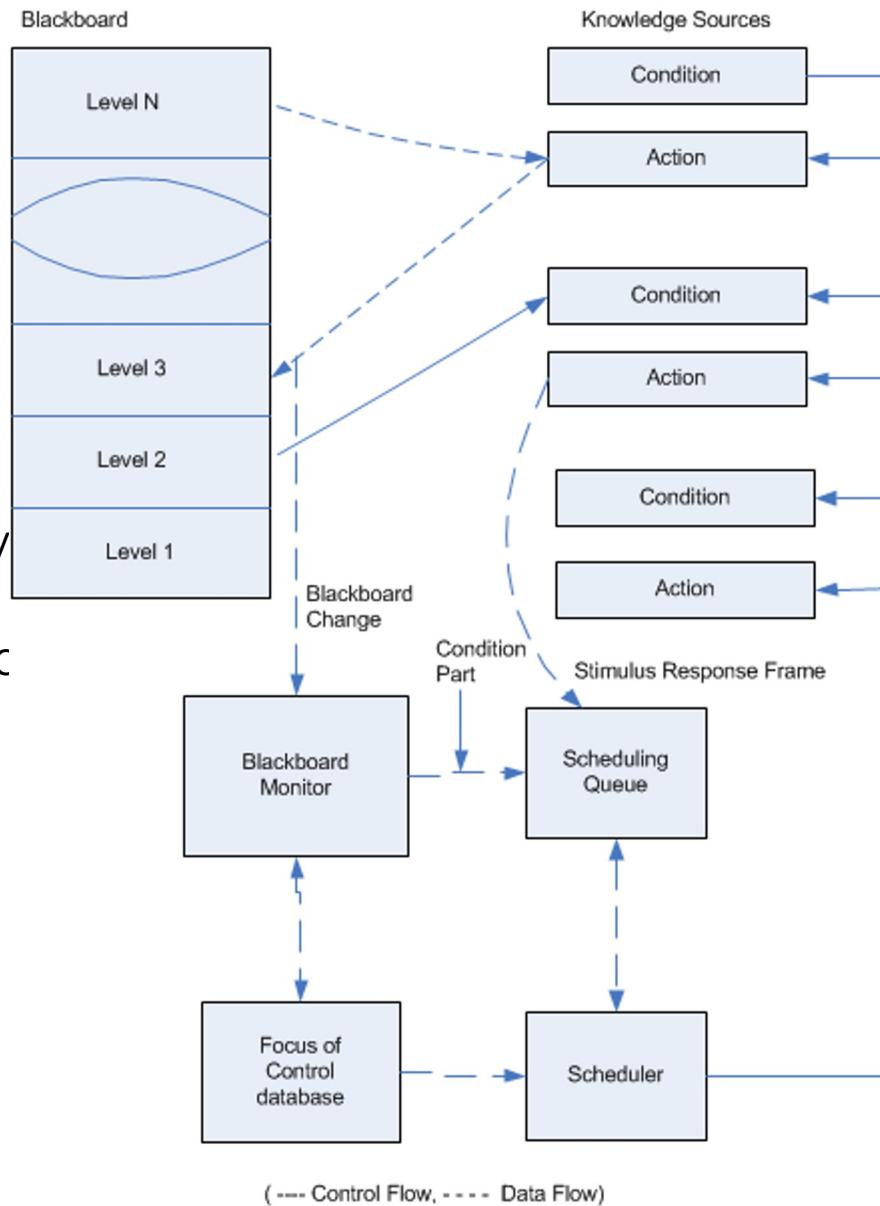


# Speech Recognition

## Blackboard

### Layers?

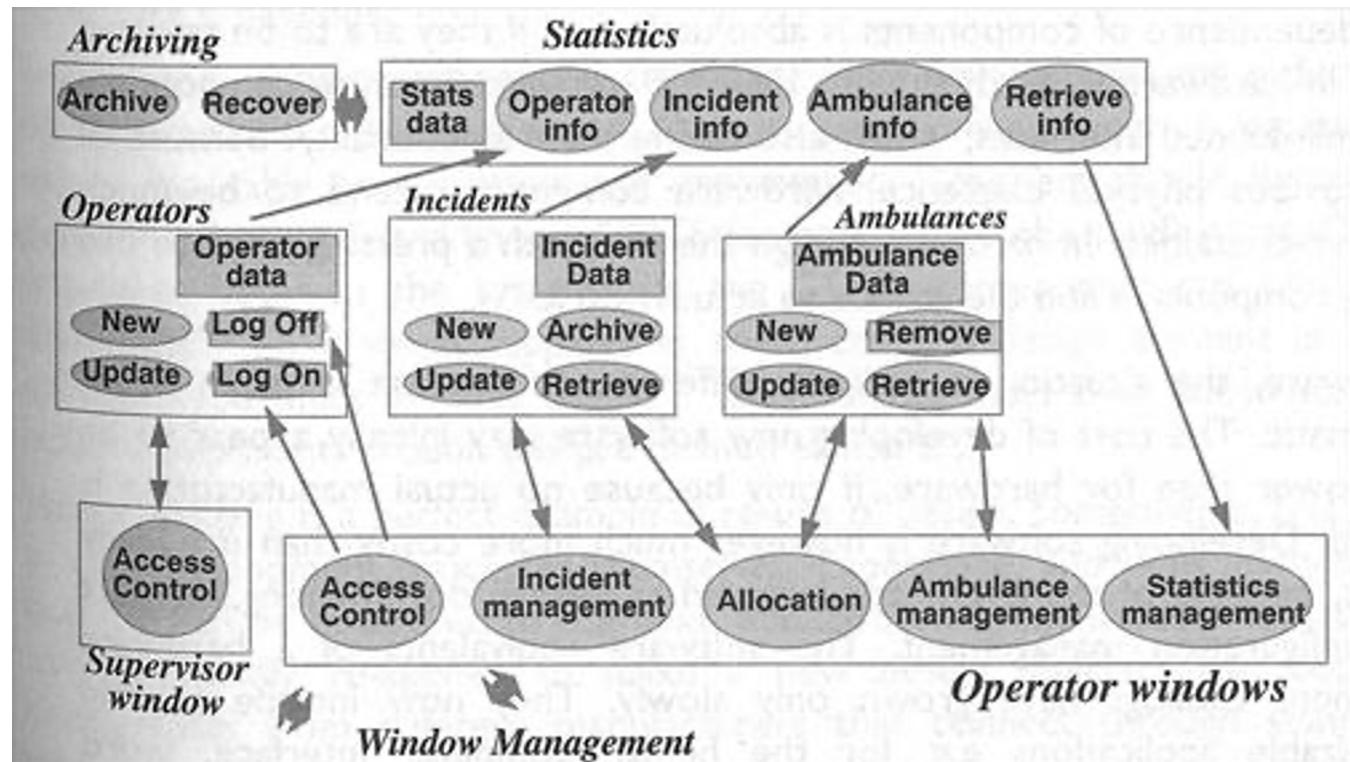
- It looks like it, but they don't appear to communicate with each other



# Ambulance Management System

Presentation  
Abstraction Control  
(different operator windows)

Shared Repository  
(statistics box)  
• It is also active?  
Maybe, but  
probably not.



# Google

Repository

- Shared?

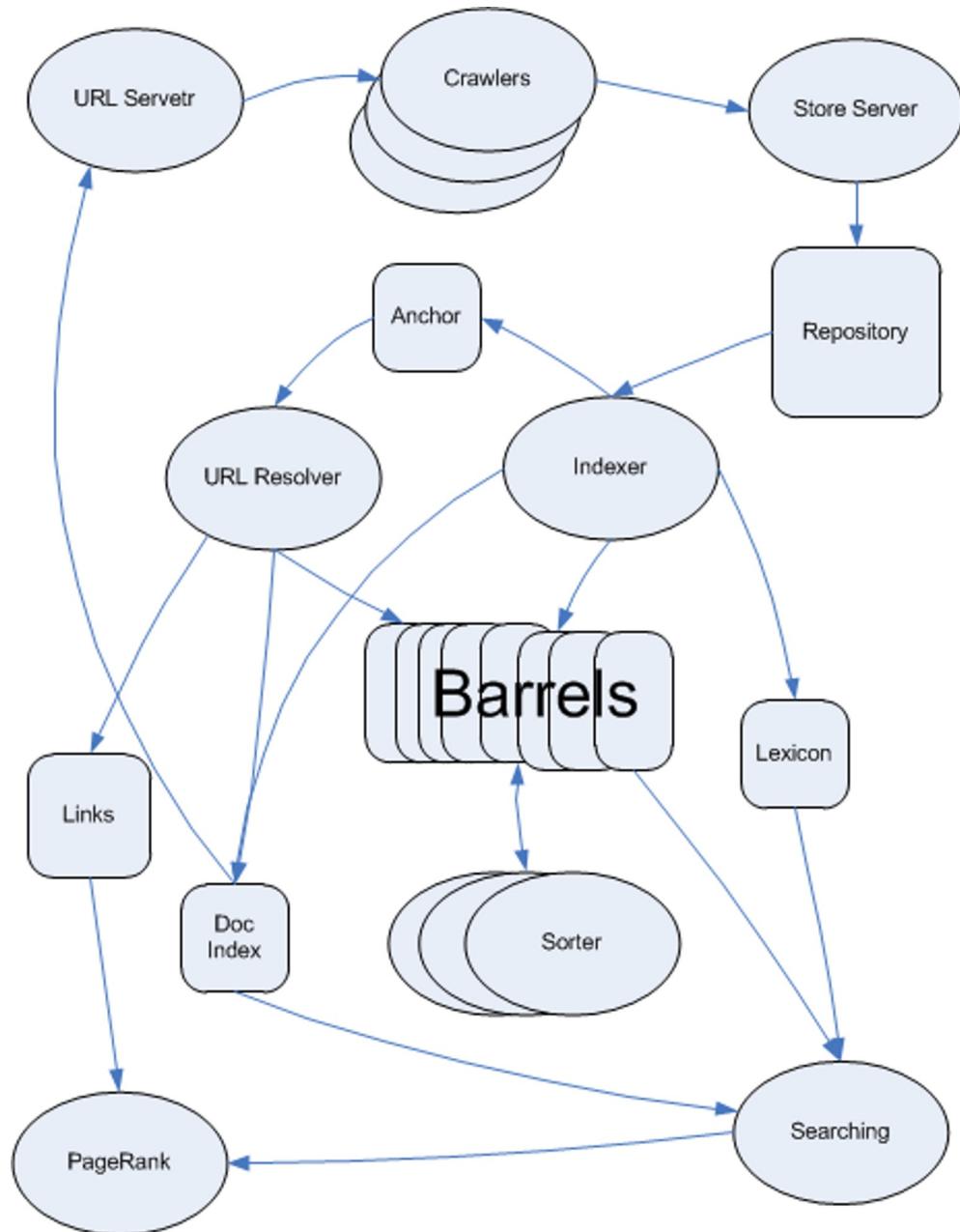
Client-Server?

Broker?

Any Layers?

What are the Barrels?

- Maybe a kind of Blackboard



# Lessons Learned

Nearly all software architectures use architecture patterns

Architects don't always use the patterns knowingly

Sometimes patterns are obvious

Sometimes not

Patterns, even accidental ones, have consequences

- More about this later

# Lessons Learned, cont.

What if we got the patterns “wrong?”

- Not what the architect intended?

One way to think of the patterns is that they describe the partitioning of the system.

- Therefore, if a pattern does describe the system’s partitioning, then it doesn’t really matter if the architect intended it or not.
- In fact, many system architectures predate the notion of architecture patterns – and we still find the patterns

# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**

# Visualizing/Documenting Architectures

“An architectural visualization defines how architectural models are depicted, and how stakeholders interact with those depictions.”

A picture or visual representation of the architecture

Pretty important: what good is an architecture if we can't see it?

There is more than one way to see an architecture

# Your Experience so far:

In the architecture documents you have seen:

What things do you like?

- Give examples
- Why do you like them?

What things do you NOT like?

- Give examples
- Why do you not like them?

# Documentation Basic Concepts

Good Architecture documentation contains both:

- Diagrams
- Text

The diagrams generally show the modules

- And the connections among the modules

The text does many things:

- Explains diagrams
- Explains tradeoffs made
- Explains rationale
- Explains the use of the system
- And more

# More basics

The goal of architecture documentation is to help people understand the architecture

Different people have different roles

- And need different views of the architecture

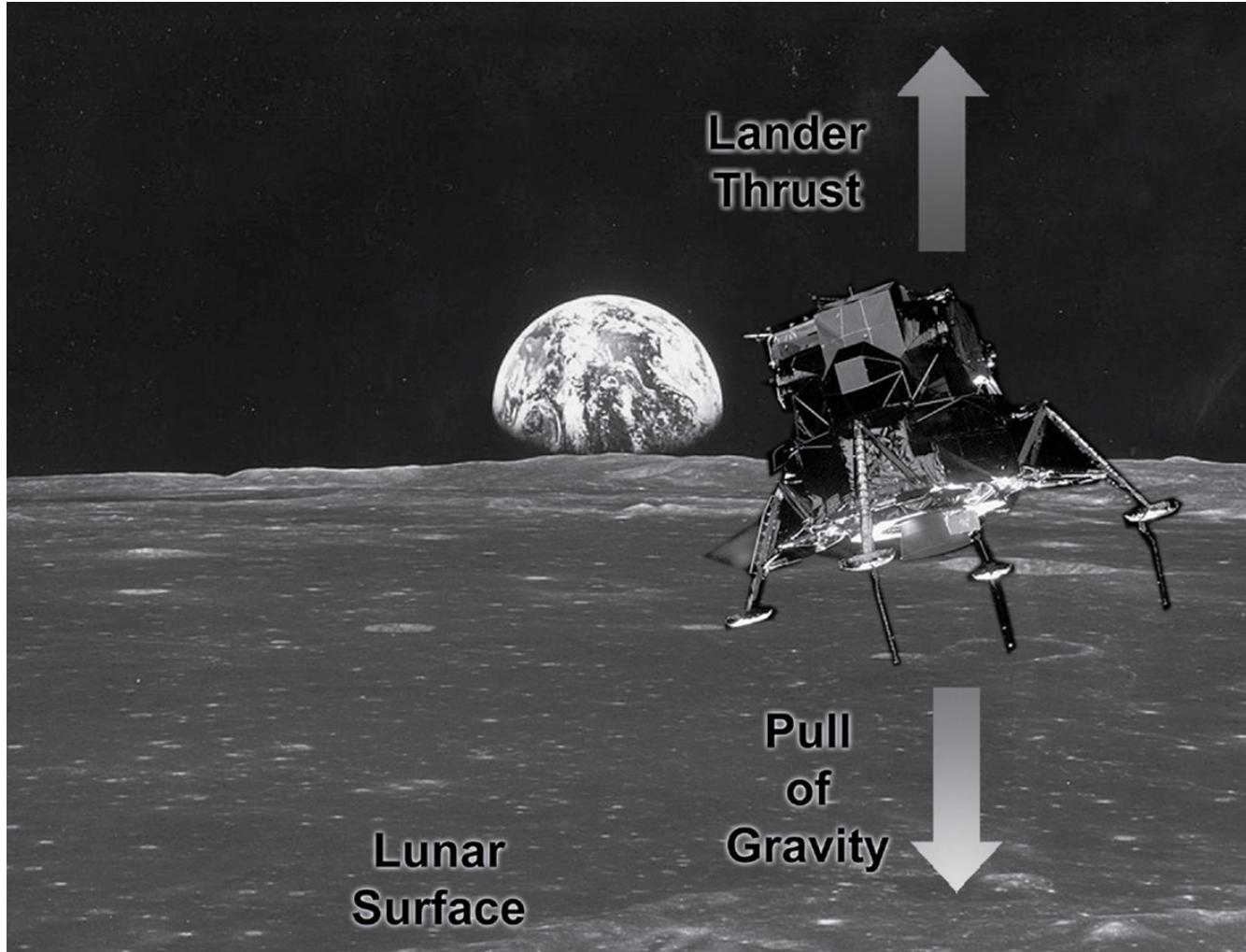
Assume that the reader does not know everything that you know about the system.

# A gallery of visualizations

Which of the following do you like, or not like?

Why?

# Lunar Lander1: Picture



# Your reaction?

Can easily see general idea

Can see some important aspects of the system  
(thrust, gravity, surface of the moon)

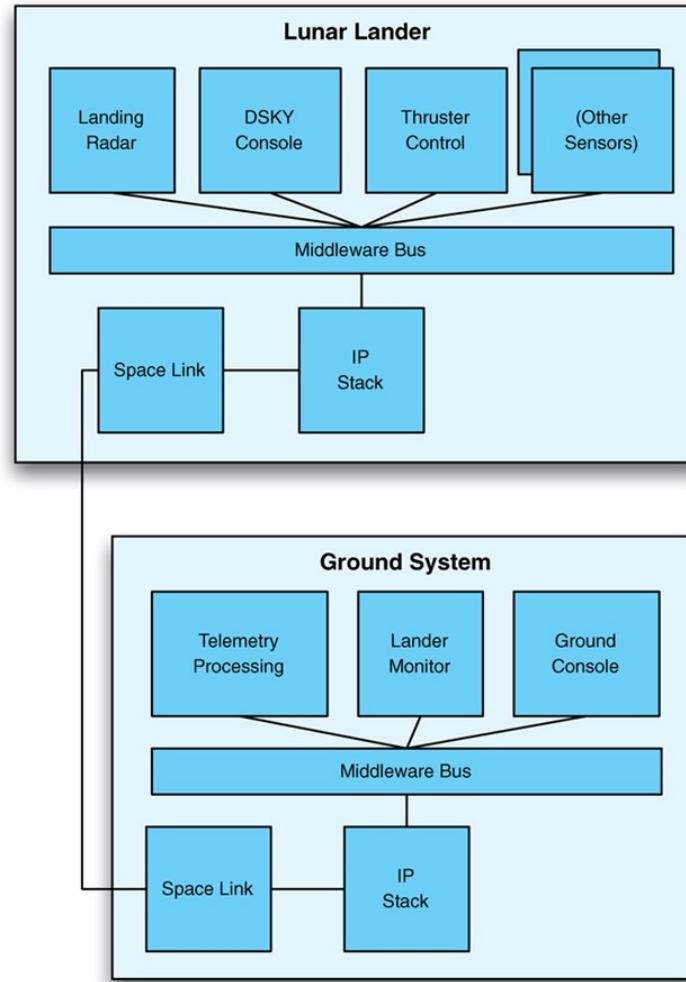
But inaccurate:

- why is the lunar lander tilted?
- May imply a three-dimensional aspect to the game

Doesn't give any info about the structure of the software

Is it useful as an implementation guide?

# Lunar Lander 2: Boxes and Lines



# Your reaction?

Gives some idea of the components

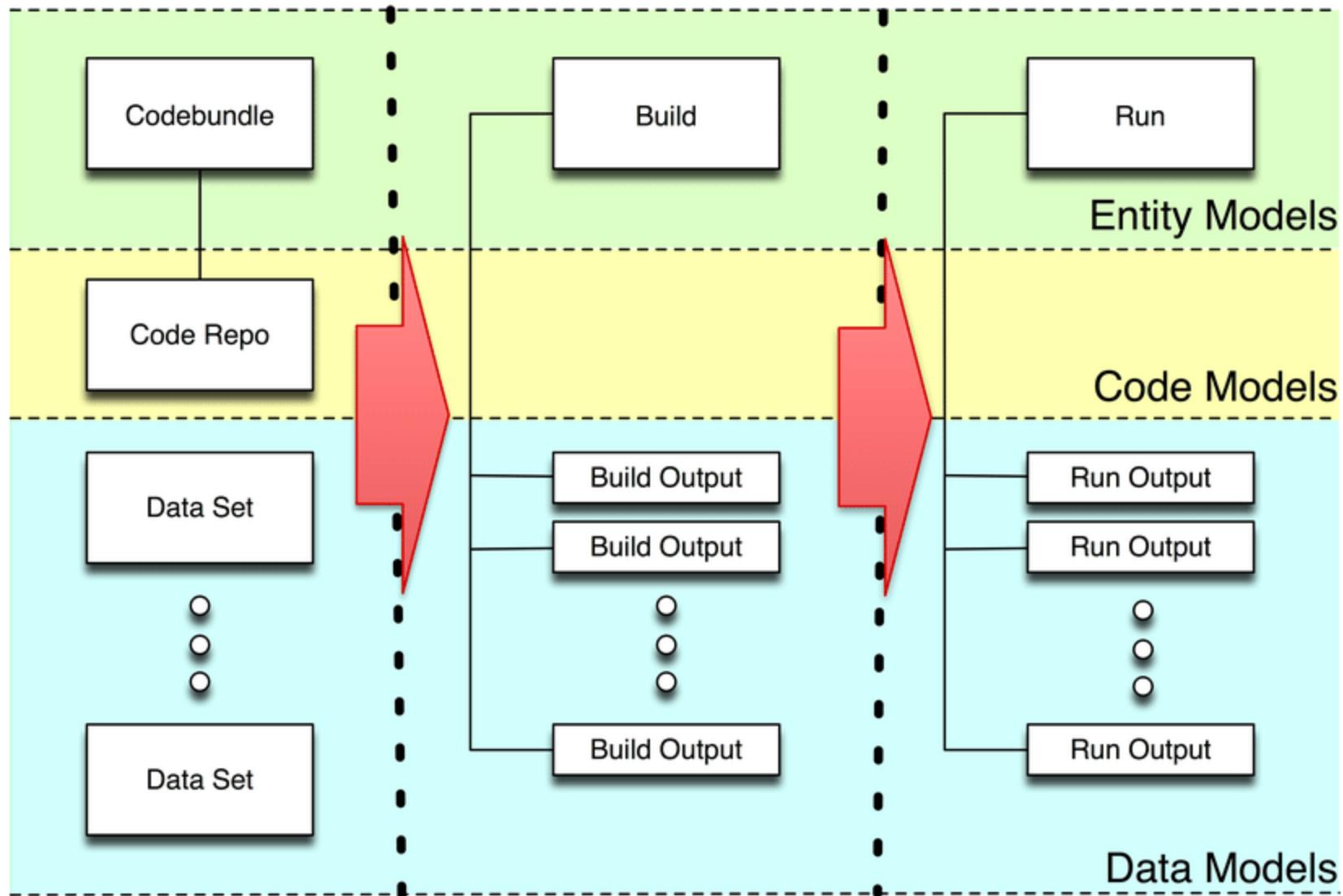
Gives some idea of how the components are connected to each other

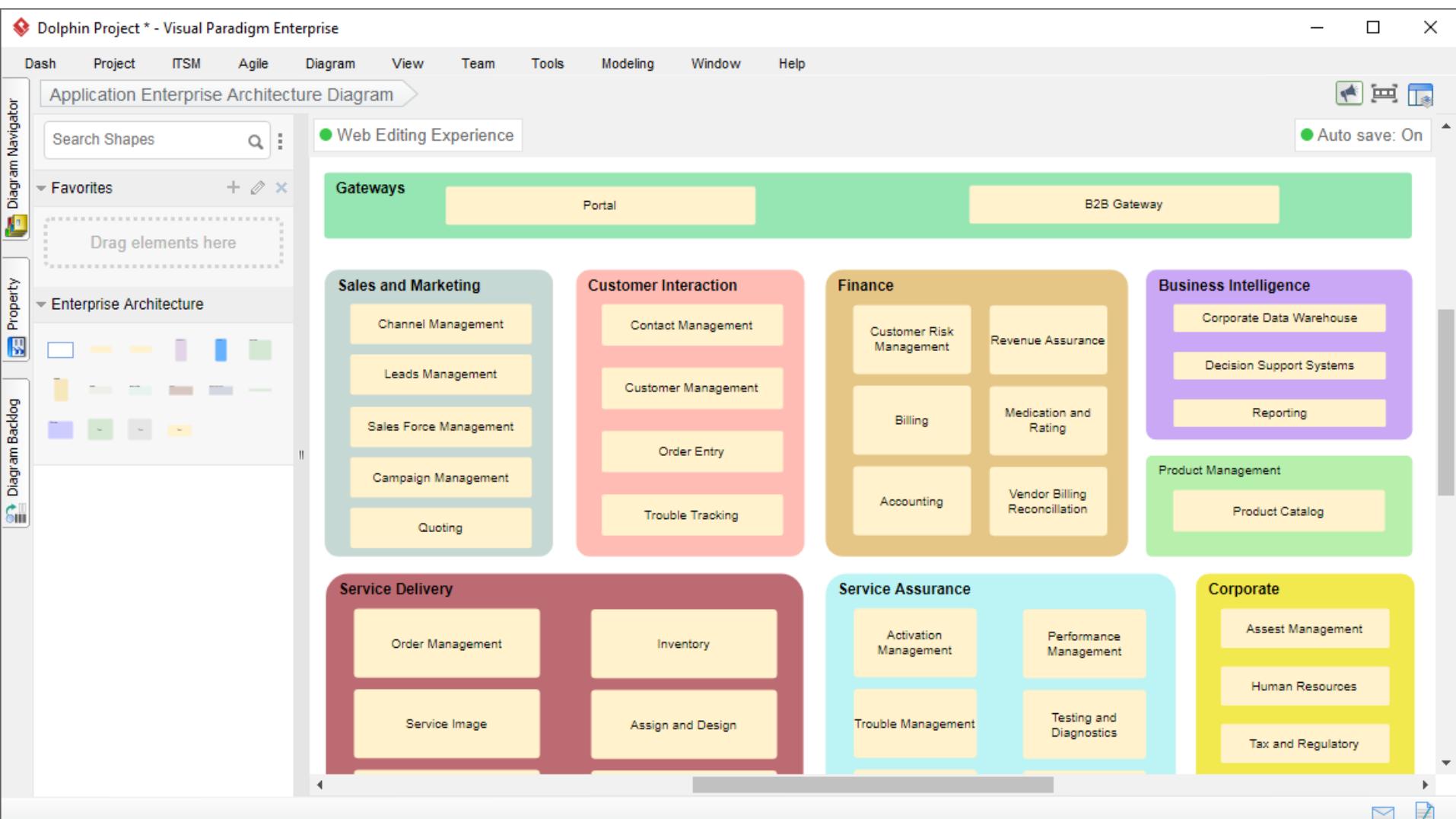
But lots of ambiguity!

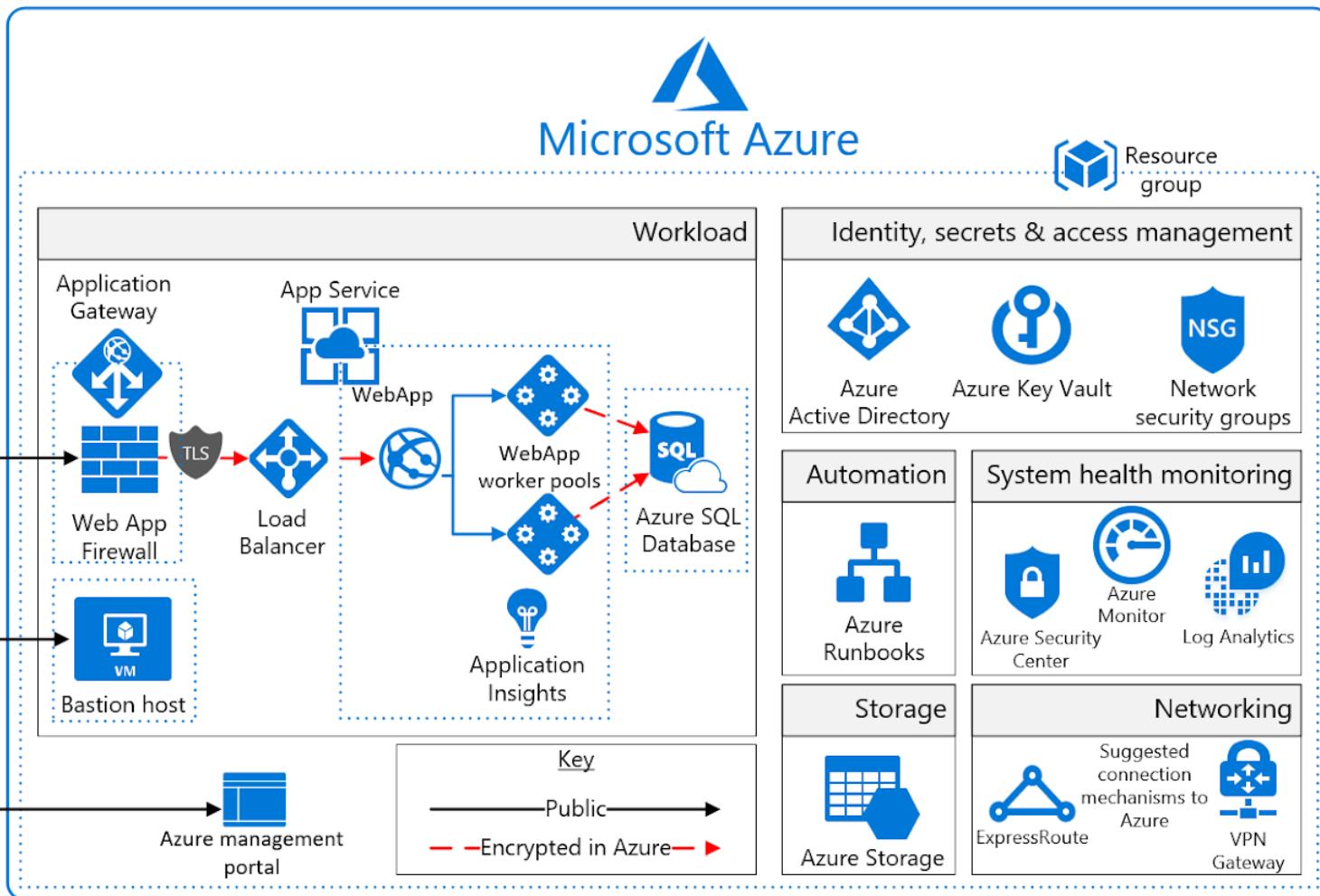
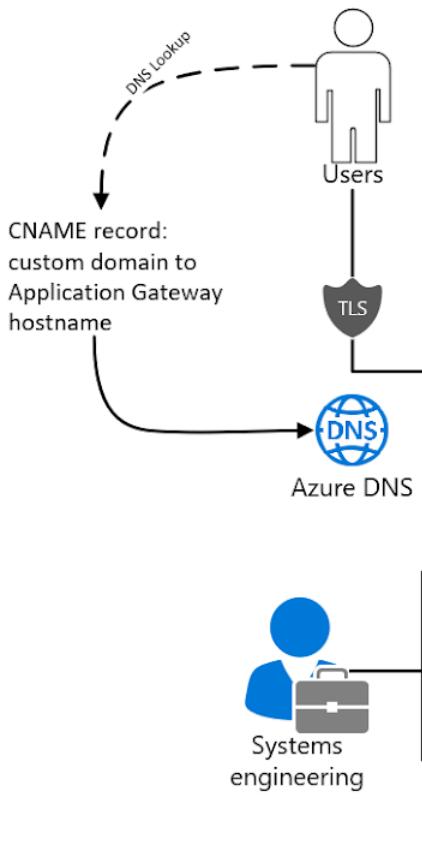
- Are all the dark blue boxes the same type of things? (um, no)
- What do the light blue boxes mean?
- What do the lines mean?

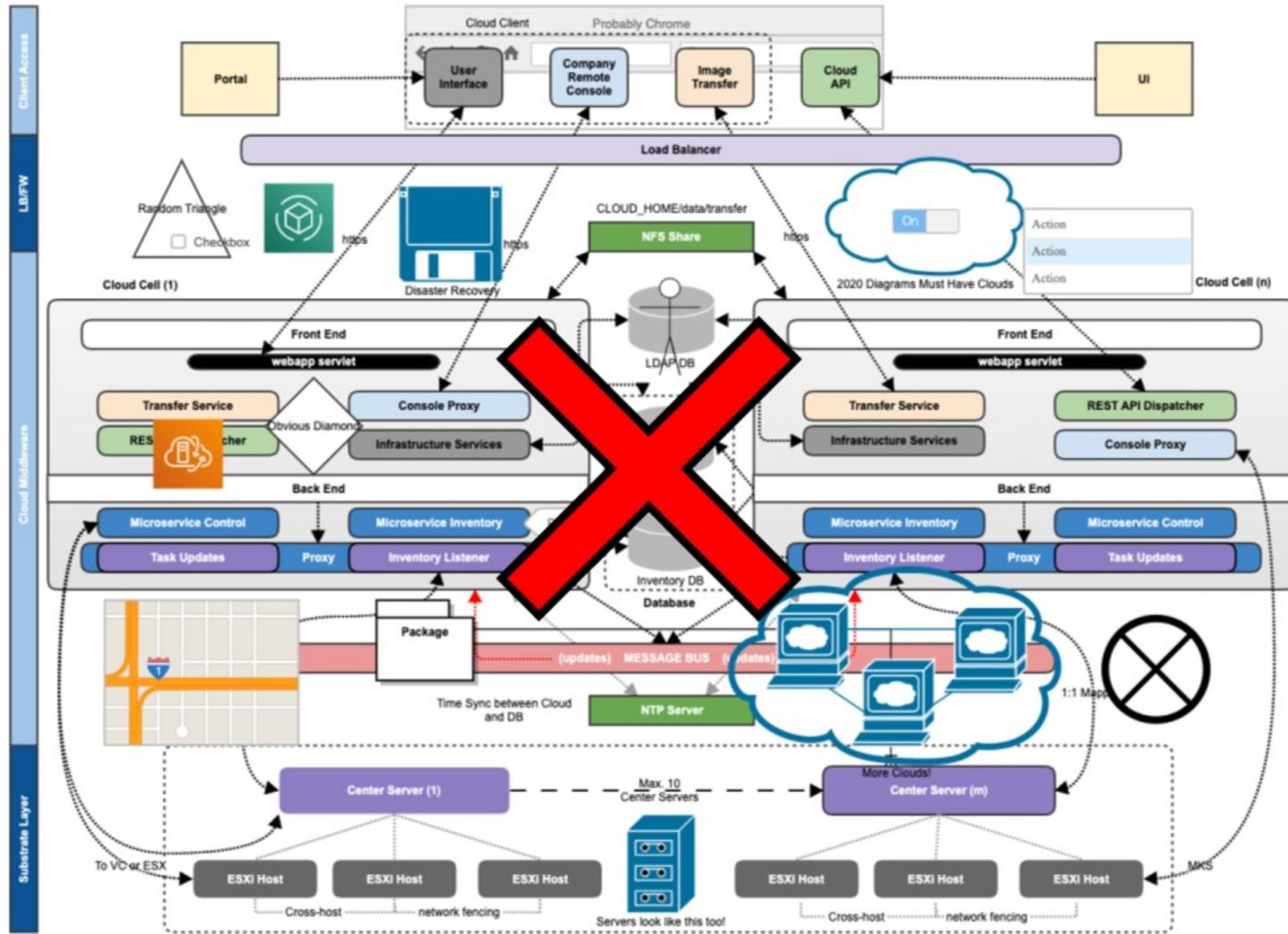
Can help you with implementation, but there are a lot of questions!

- Lots of room for misinterpretations









# Textual: xADL (an architectural description language)

```
<instance:xArch xsi:type="instance:XArch">
  <types:archStructure xsi:type="types:ArchStructure"
    types:id="ClientArch">
    <types:description xsi:type="instance:Description">
      Client Architecture
    </types:description>
    <types:component xsi:type="types:Component"
      types:id="WebBrowser">
      <types:description xsi:type="instance:Description">
        Web Browser
      </types:description>
      <types:interface xsi:type="types:Interface"
        types:id="WebBrowserInterface">
        <types:description xsi:type="instance:Description">
          Web Browser Interface
        </types:description>
        <types:direction xsi:type="instance:Direction">
          inout
        </types:direction>
      </types:interface>
    </types:component>
  </types:archStructure>
</instance:xArch>
```

```
xArch{
  archStructure{
    id = "ClientArch"
    description = "Client Architecture"
    component{
      id = "WebBrowser"
      description = "Web Browser"
      interface{
        id = "WebBrowserInterface"
        description = "Web Browser Interface"
        direction = "inout"
      }
    }
  }
}
```

# Your reaction?

Precise

Can use tools to validate it, and to maybe generate some code

Not very easy to understand!

Can anyone tell me the structure of the architecture we just looked at?

NOTE #1: I have not seen something like this in practice.

Note #2: It is fundamentally WRONG!

- It is precise, but software architecture is **by definition imprecise**

# **What makes a good visualization?**

(By extension, what are characteristics of a good visualization tool?)

# Fidelity

How faithfully does a view represent the underlying model?

Everything displayed should be absolutely correct  
(matches the model)

But not everything in the model must appear in the model

- In fact, it's often a good idea to show only certain aspects of the model

Note that fidelity affects both depiction and interaction

- For example, some visualization tools may allow the user to make changes

# Consistency

Are similar concepts displayed in a consistent manner?

Is a visualization mechanism (e.g., a box) used for different concepts?

In other words, is the visualization consistent with itself? (internal consistency)

Example, UML:

- UML always depicts an object as a rectangle with an underlined name
- But a dashed open-headed arrow means “dependency” in most UML diagrams, but it means “asynchronous invocation” in a UML sequence diagram

Note: there is a balance: extreme consistency can lead to a huge and confusing variety of symbols

# Comprehensibility

VERY important

Should be able to understand it rather quickly

Notation, diagrams should be intuitive

Helps to keep scope of the visualization narrow

Don't try to display too much information at once

Note the people may bring assumptions about what symbols may mean: don't violate them



# Dynamism

How well does the visualization support models that change over time

Information flows two ways:

- Change the model → change the visualization
- Change the visualization → change the model

Closely related to the tool(s) used

Note: pretty simple if the model has a single integrated visualization. Not so simple if there are multiple visualizations of the model.

# View Coordination

Multiple views of a model must be consistent with each other

Note that a person might use multiple views simultaneously.

# Aesthetics

Yes, it should look good

Lots of material available on how to make things look good

Yes, it is very subjective

Doesn't seem important, but it might make a difference in acceptance by potential users

(See materials by Edward Tufte)

# Edward Tufte (quick summary)

“The Visual Display of Quantitative Information”

- Lessons apply to architecture diagrams too

Parsimony:

- Data-ink ratio
- Avoid chartjunk (distractions: focus on the information)
- Does NOT mean minimizing information

Envisioning information:

- 3-dimensional depictions
- Layering (multiple depictions that can overlay on each other)
- Time variation (different depictions over time)
- Small multiples (many small depictions of related data)

# **Extensibility**

How easy is it to modify a visualization?

Tools can help (or not) (e.g. drawing tools)

Ability to add new capabilities

# Grok Factor

(English geek-slang)

When you look at a visualization, how long does it take you to understand it?

It should be short

# Constructing a Visualization

Borrow elements from Similar Visualizations

- The idea: if you use a common shape, the reader should immediately know what it stands for

Be consistent among visualizations

Give meaning to each visual aspect of elements

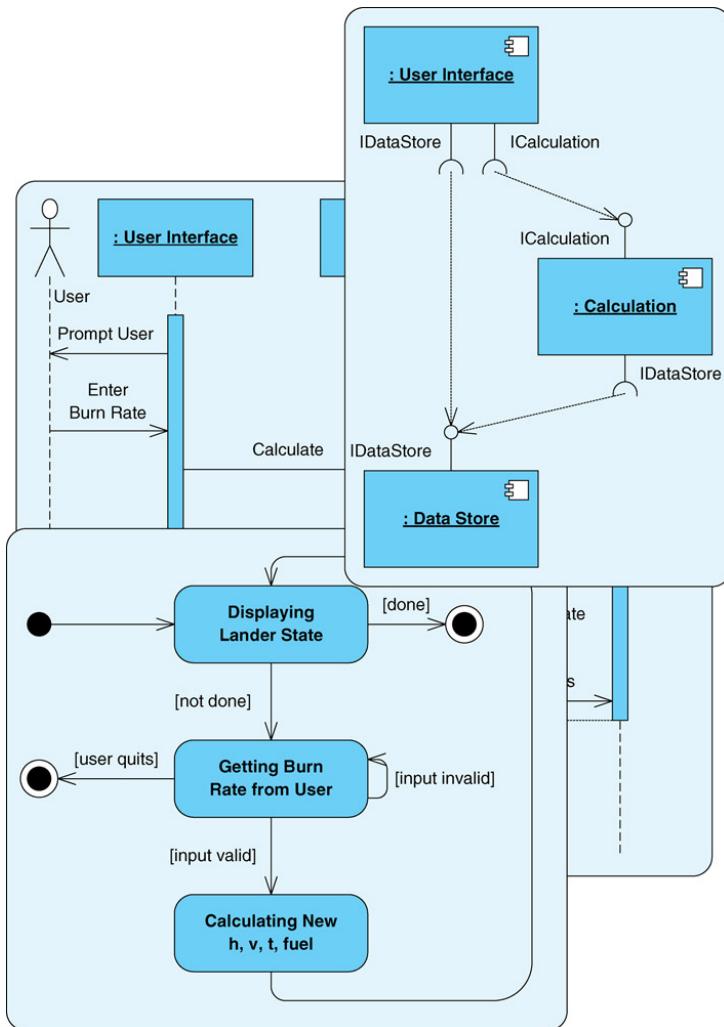
- For example, if you use colors

Document the meaning of visualizations

Balance traditional and innovative (user) interfaces

- E.g., traditional look and feel: PowerPoint, Visio

# **Example: different views in UML**



# Options

Use existing architecture visualization tools/notations

- UML is the most standard
- It's kind of heavy

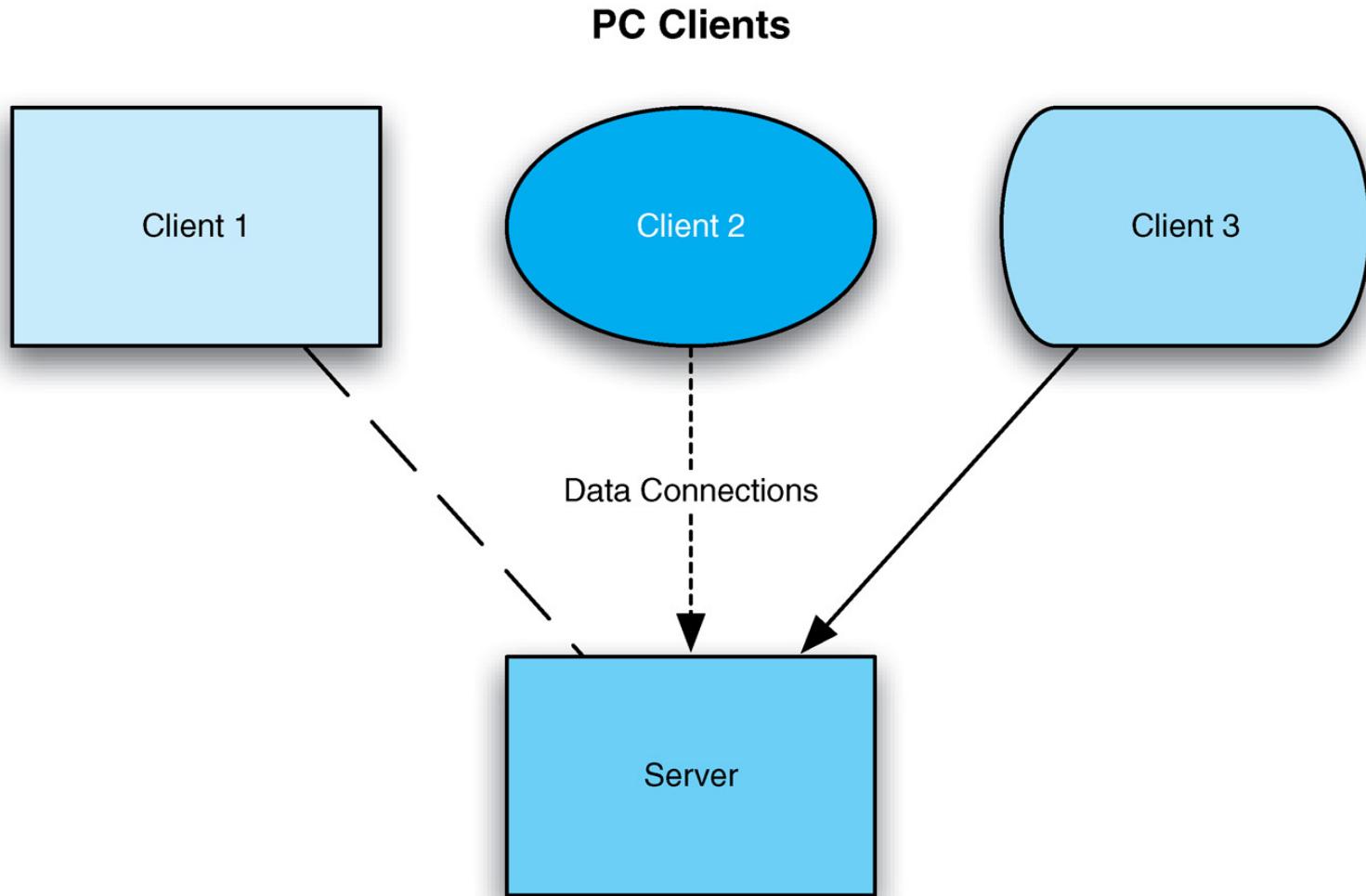
Make up your own diagrams

- Often easier
- But can easily lead to ambiguity and other problems!
- **My Experience: Nearly everybody makes up their own styles**

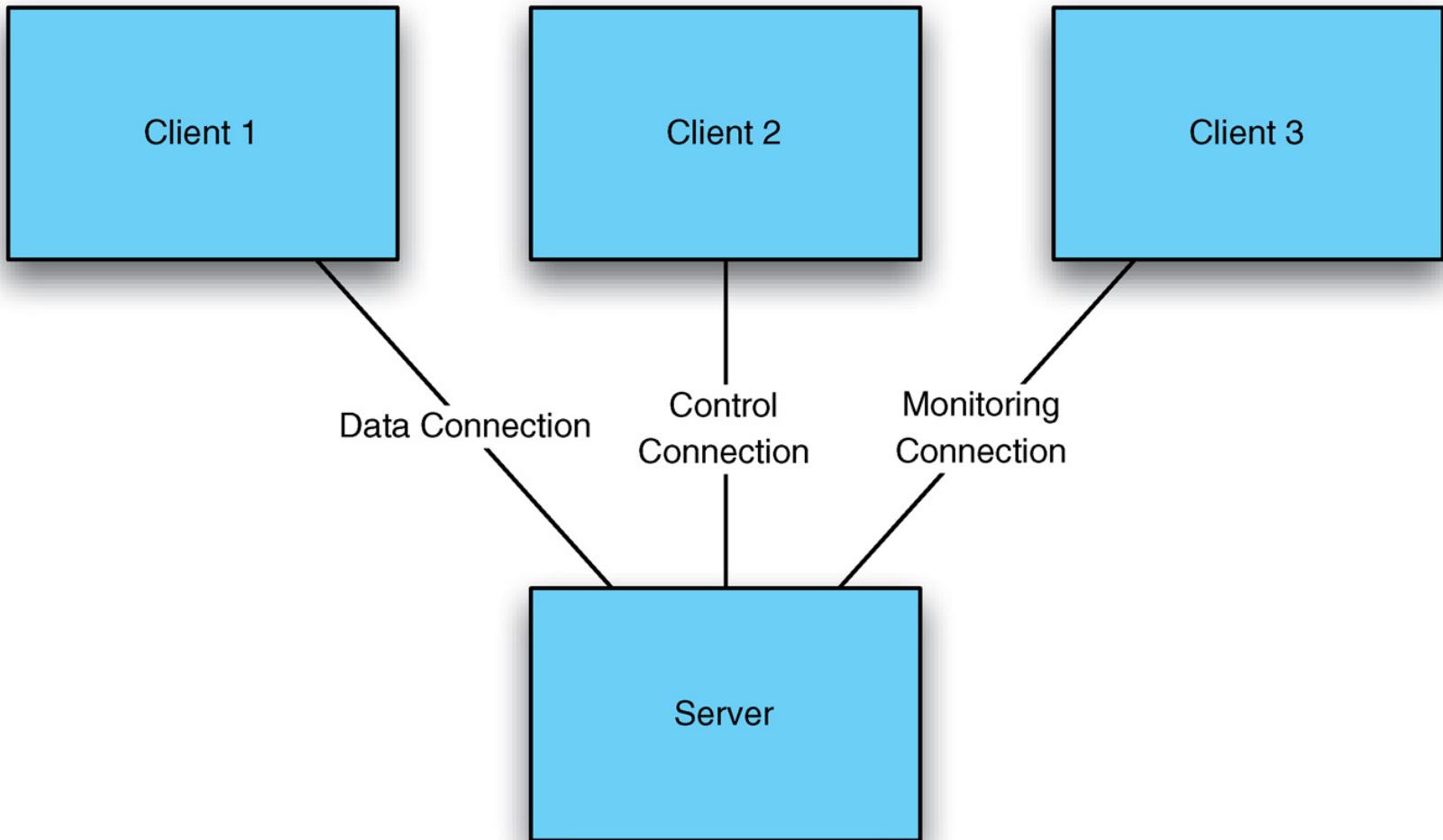
Common Issues (see next slides)

- Mostly in the “make your own” styles
- But can appear even with the use of standard notations

# What is wrong with this picture?



# What is wrong with this picture?



Notes on previous slide:

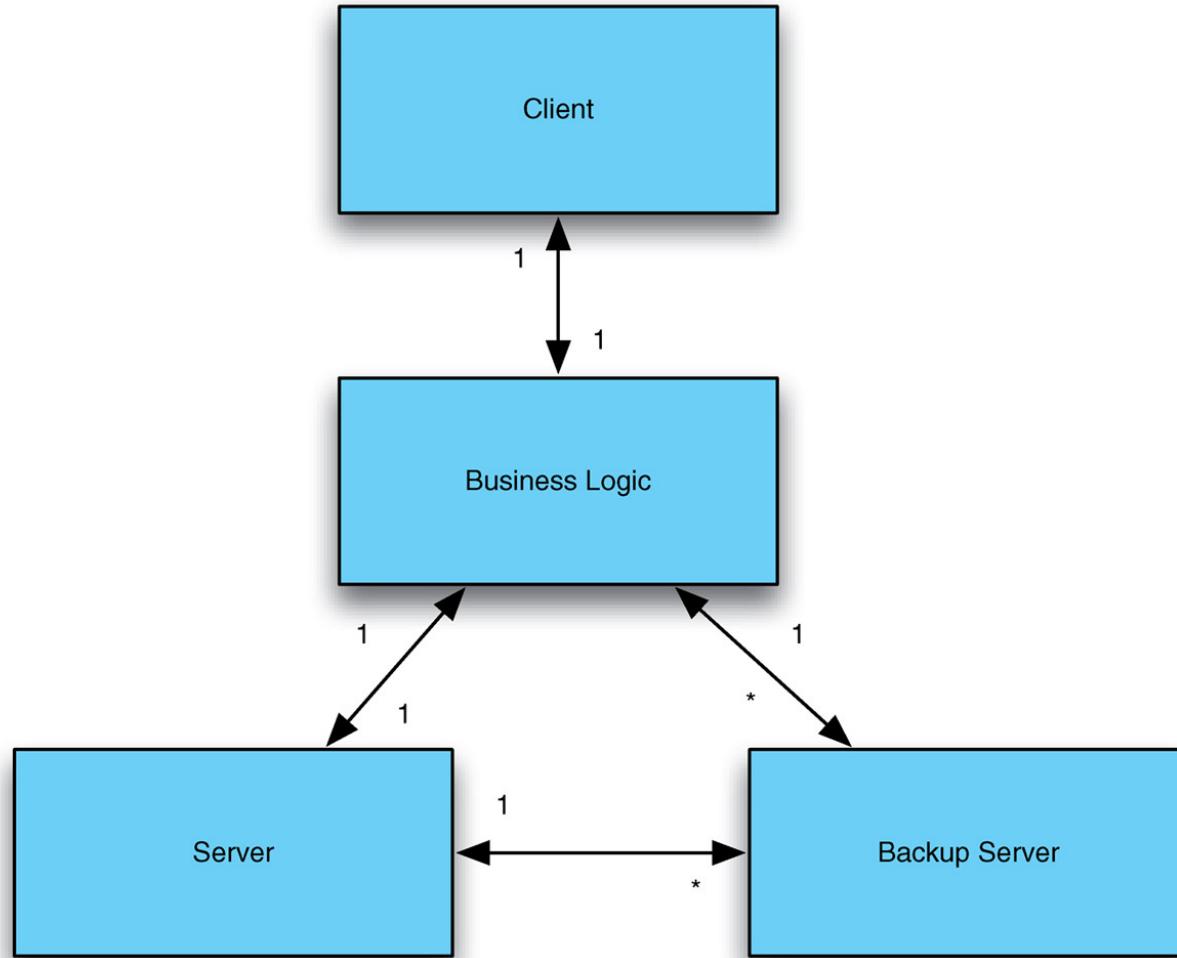
Clients and Server have the same size and color boxes

- That might be ok.
- But they need different labels, of course

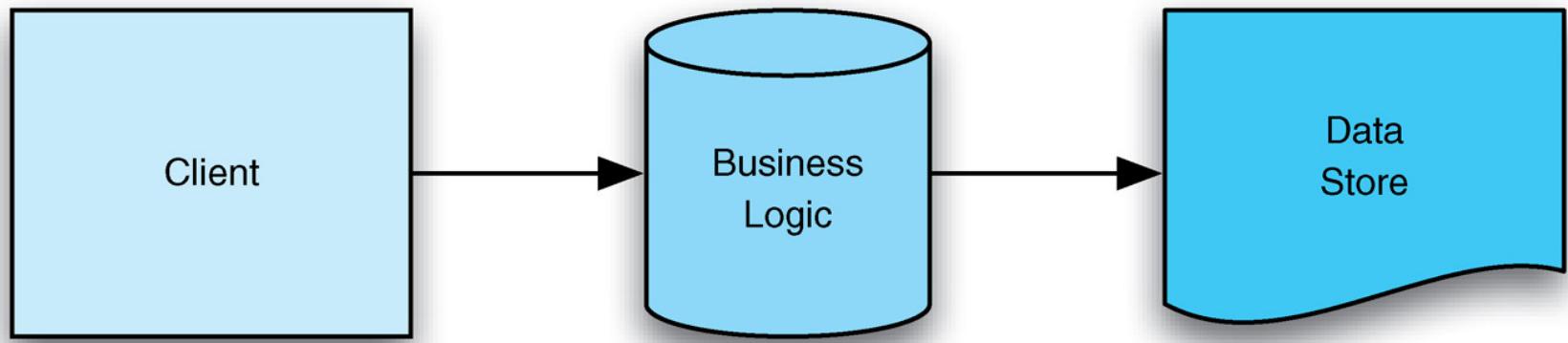
Different types of connectors use the same thing: lines

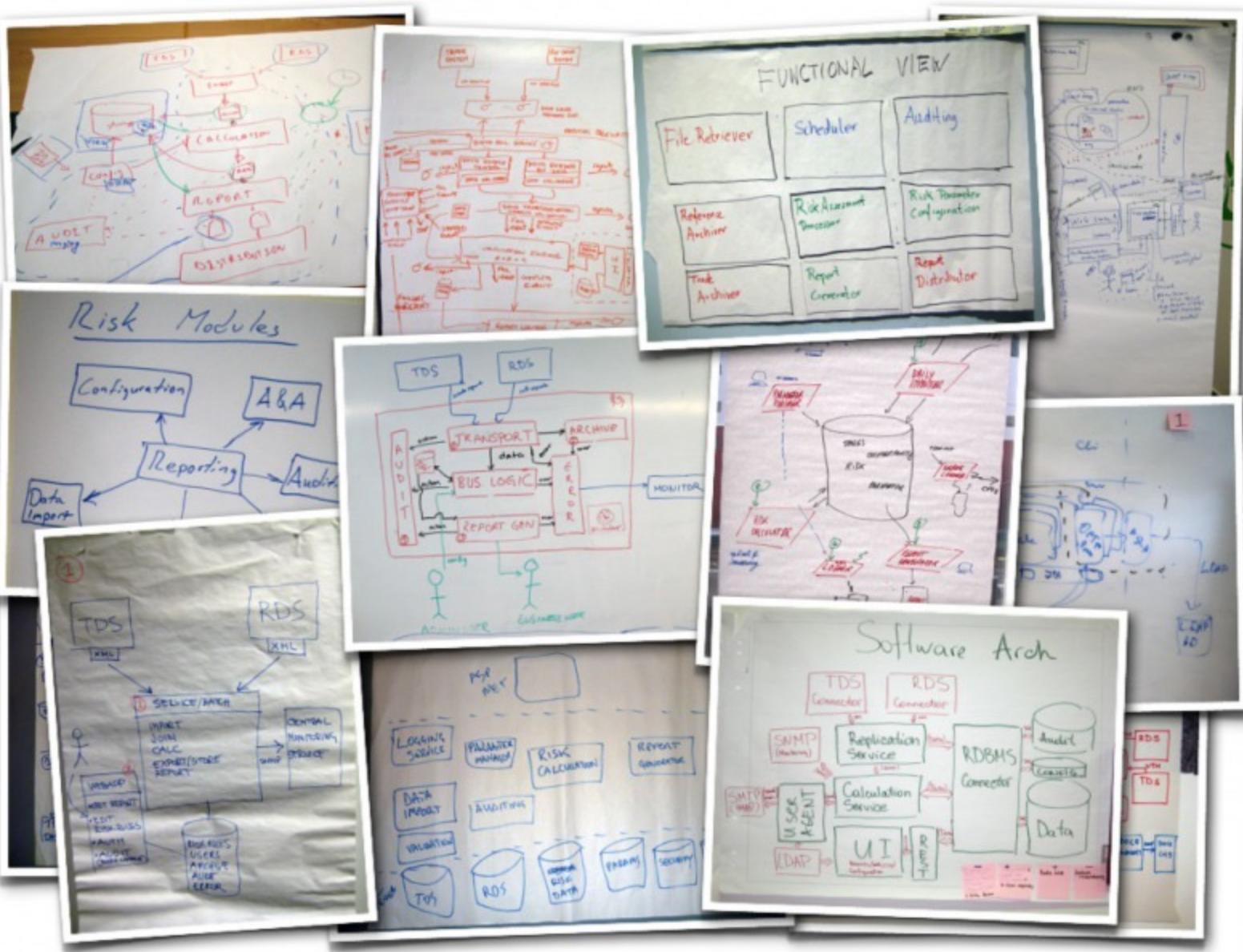
- That might also be ok.
- If you have different types of lines, you need a legend, and it gets messy.
- It may be clearest just to have labels and explanations.

# Issue: decorations without meaning



# What is wrong with this picture?





# Comments about the previous slide

Color-coding is not explained or is inconsistent.

The purpose of diagram elements (i.e. different styles of boxes and lines) is not explained.

Key relationships between diagram elements are missing or ambiguous.

Generic terms such as “business logic”, “service” or “data access layer” are used.

Technology choices (or options) are omitted.

Levels of abstraction are mixed.

Diagrams lack context or a logical starting point.

And yet, there is a lot of information there...

- But do we get it right?

Source: <https://www.voxxed.com/2014/10/simple-sketches-for-diagramming-your-software-architecture/>

# But ...

The diagrams were probably VERY useful to the team who drew them up:

- They probably know what the colors mean
- They probably have all the necessary context
  - (in their heads!)
- It's probably perfectly clear to them
- → They probably drew them while they were designing the architecture!
- → In other words, it is probably a decent model.

The trick is to make the diagrams meaningful to  
**other people**

- You probably must redo your initial diagrams (and add the information that's in your head)

# Theory vs. Practice Alert!

Next slides show the intention

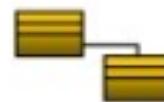
# “Official” approach

4+1 view model architecture  
*(for distributed systems)*

- 1. Logical View (or Structural View)** - *an object model of the design*
- 2. Process View (or Behavioral View)** - *concurrency and synchronization aspects*
- 3. Development View (or Implementation View)** - *static organization (subset) of the software*
- 4. Physical View (or Deployment View)** - *mapping of the software to the hardware*
- +1. Use-cases View ( or Scenarios)** - *various usage scenarios*

## 1. Logical View = (The Object-oriented Decomposition)

- **Viewer:** End-user
- **Considers:** **Functional Requirements-** What the system should provide in terms of services to its users.
- **What this view shows?**
  - *"This view shows the components (objects) of the system as well as their interaction/relationship".*
  - Notation: Object and dynamic models
  - **UML diagrams:** Class, Object, State Machine, Composite Structure diagrams



## 2. Process View = (The Process Decomposition)

- **Viewer:** Integrator(s)
- **Considers:** Non-Functional Requirements - (concurrency, performance, scalability, usability, resilience, re-use, comprehensibility, economic and technology constraints, trade-offs, and cross-cutting concerns - like security and transaction management)
- **What this view shows?**
  - *"This view shows the processes (workflow rules) of the system and how those processes communicate with each other".*
  - **UML diagrams:** Sequence, Communication, Activity, Timing, Interaction Overview diagrams



### 3. Development/Implementation View

= (The Subsystem Decomposition)

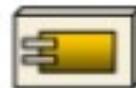
- **Viewer:** Programmers and Software Managers
- **Considers:** **Software module organization** (Hierarchy of layers, software management, reuse, constraints of tools)
- **What this view shows?**
  - *"This view shows the building blocks of the system".*
- **UML diagrams:** Component, Package diagrams
  - Package Details, Execution Environments, Class Libraries, Layers, Sub-system design



## 4. Physical/Deployment View

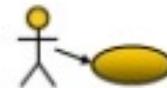
= (Software to Hardware Mapping)

- **Viewer:** System Engineers
- **Considers:** Non-functional Requirements for hardware  
(Topology, Communication)
- **What this view shows?** Non-functional
- *"This view shows the system execution environment".*
- **UML diagrams:** Deployment diagrams
- **Non-UML diagrams:** Network Topology (not in UML)



## 5. Use-case View/Scenarios = (putting it altogether)

- **Viewer:** All users of other views and Evaluators
- **Considers:** System consistency, validity
- **What this view shows?**
- *"This view shows the Validation and Illustration of system completeness. This view is redundant with other views."*
- **UML diagrams:** Use-case diagram, User stories



## Relationships between Views

- The *Logical View* and the *Process View* are at a **conceptual level** and are used from analysis to design.
- The *Development View* and the *Deployment View* are at the **physical level** and represent the actual application components built and deployed.
- The *Logical View* and the Development View are tied closer to functionality (**functional aspect**). They depict how functionality is modeled and implemented.
- The *Process View* and Deployment View realizes the **non-functional aspects** using behavioral and physical modeling.
- *Use Case View* leads to **structural elements** being **analyzed** in the Logical View and **implemented** in the Development View. The scenarios in the Use Case View are **realized** in the Process View and **deployed** in the Physical View.

## Why is it called the 4 + 1 instead of just 5?

- **The Use-case View:** The use case view has a special significance. Views are effectively redundant (i.e. Views are interconnected).
  - However, all other views would not be possible without use case view.
  - It details the high levels requirements of the system.
  - The other views detail how those requirements are realized.

# The Real World

## Common Practices:

- People rarely use all five views
  - Plus Use Cases/user stories
    - User stories more common
- Put logical and physical views in the same picture
- Dynamic views often neglected
- Connectors are usually poorly documented

## Recommendations

- At least one static view, usually at least two
- Usually separate the physical and logical views
- Dynamic view is usually important
- Favor use cases over user stories
- Make connectors clear
- Notation (will cover that later)

# “View”: One more thing

“View” doesn’t equate to “Diagram”

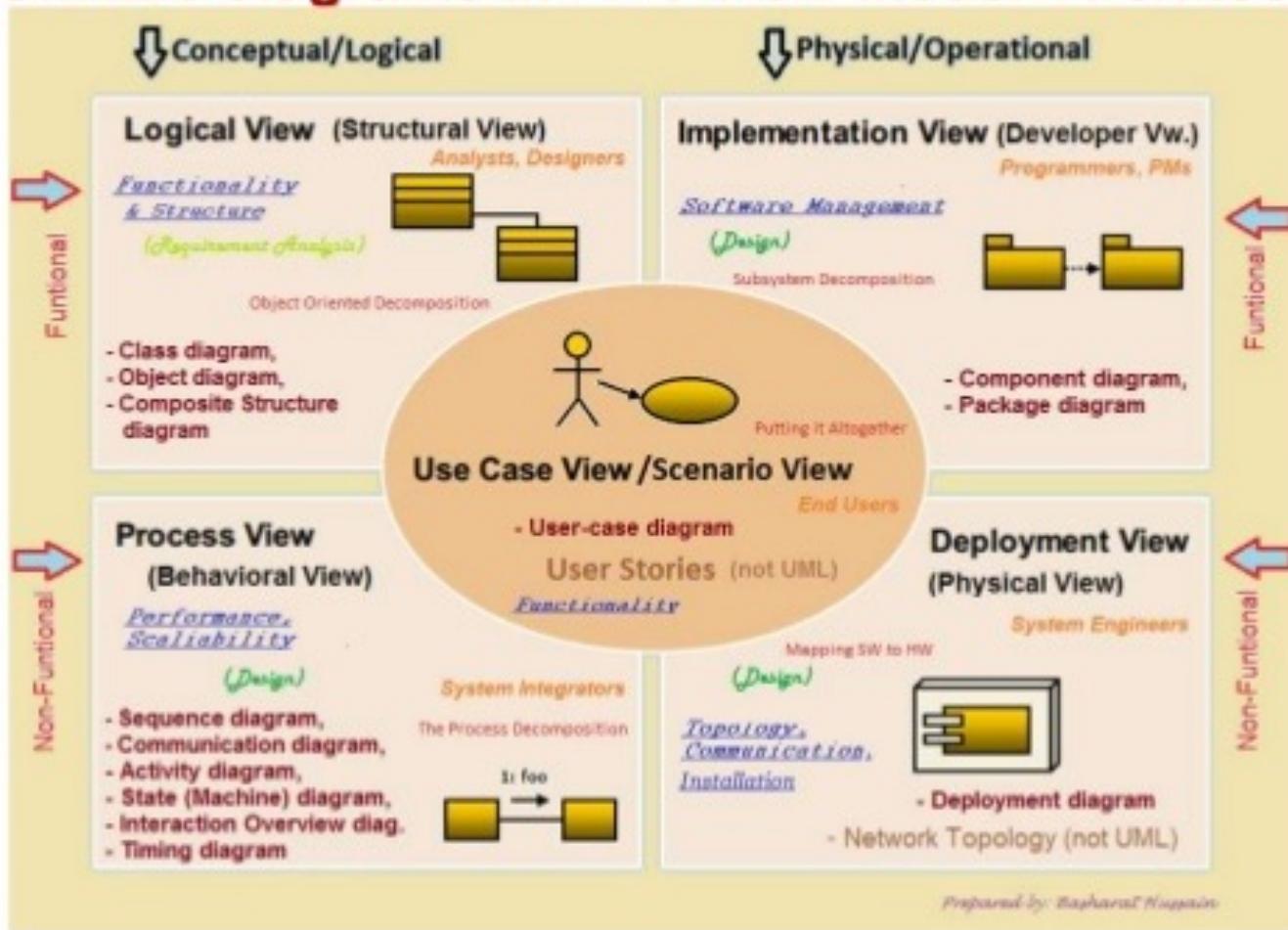
- A “View” is a “point of view”
- In other words: how someone looks at the architecture

Therefore:

- One view may have several diagrams
- A view may not have any diagrams at all
  - (Ok, that is rare)
  - Consider the “Deployment View”. Sometimes it is just a list of instructions in text.

# Common: 4+1 Views

## 13 UML2.0 diagrams in '4+1 View Model' Architecture

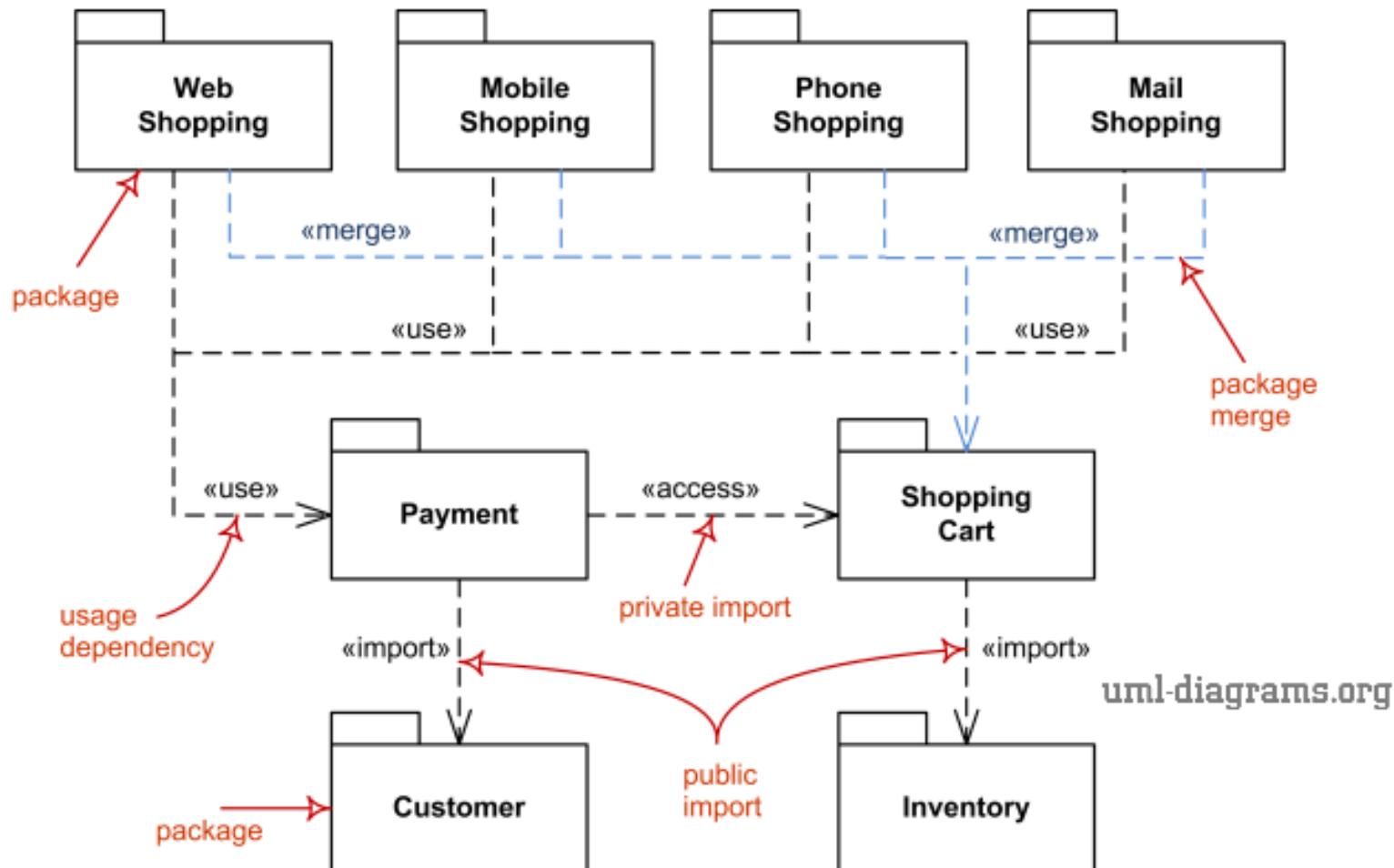


# Notation

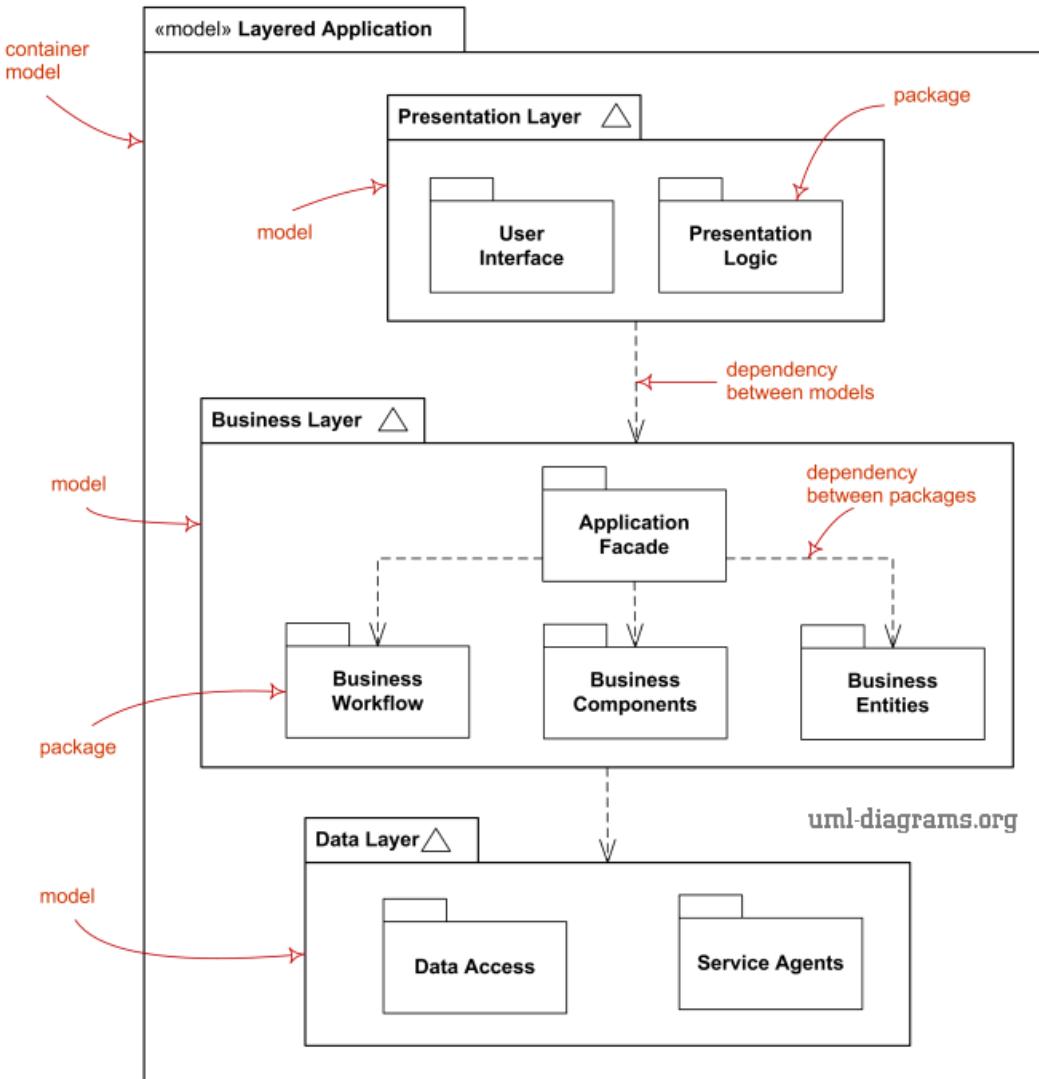
The next slides show UML notation

Note that not all UML is used commonly

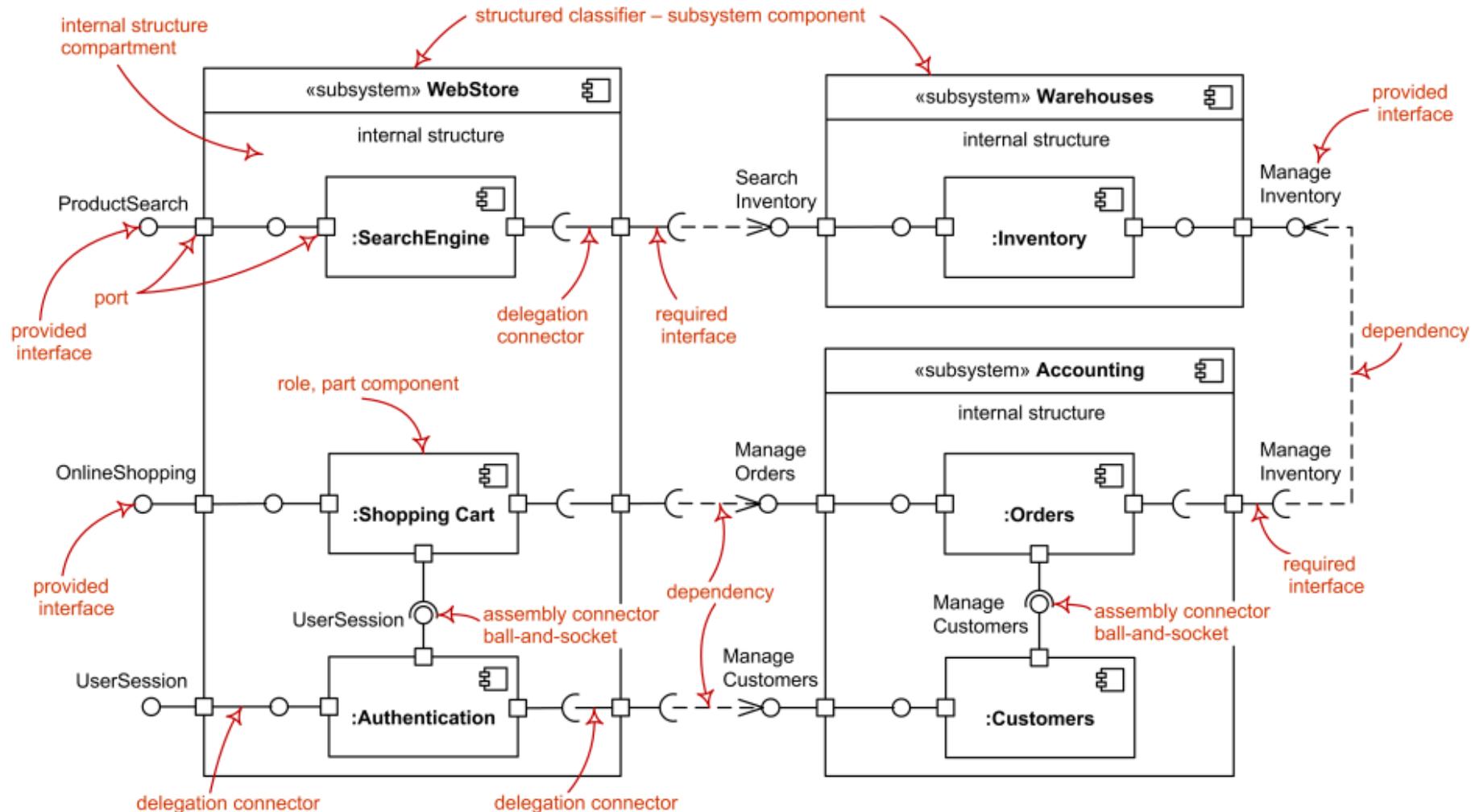
# UML Package Diagram



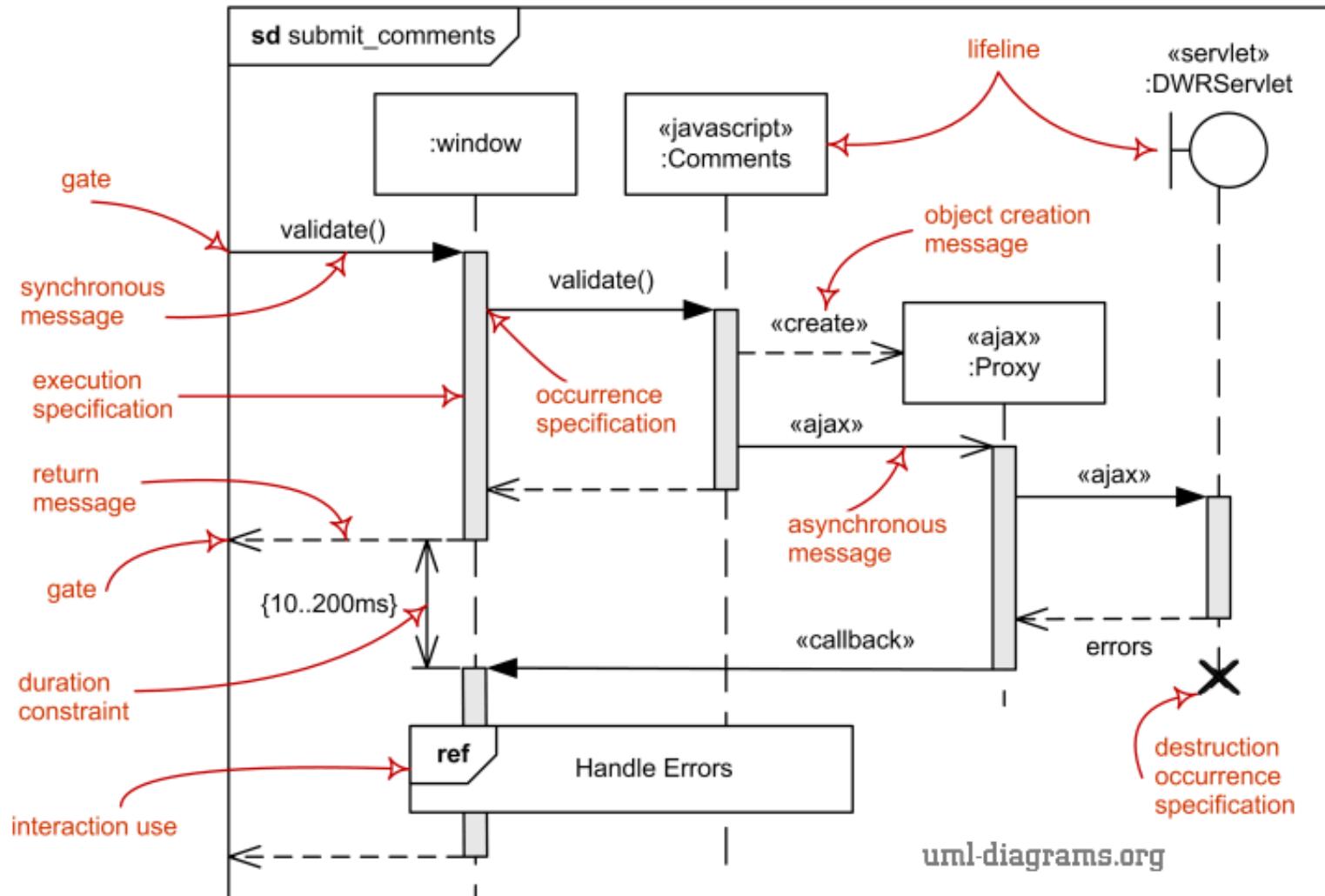
# UML Model diagram



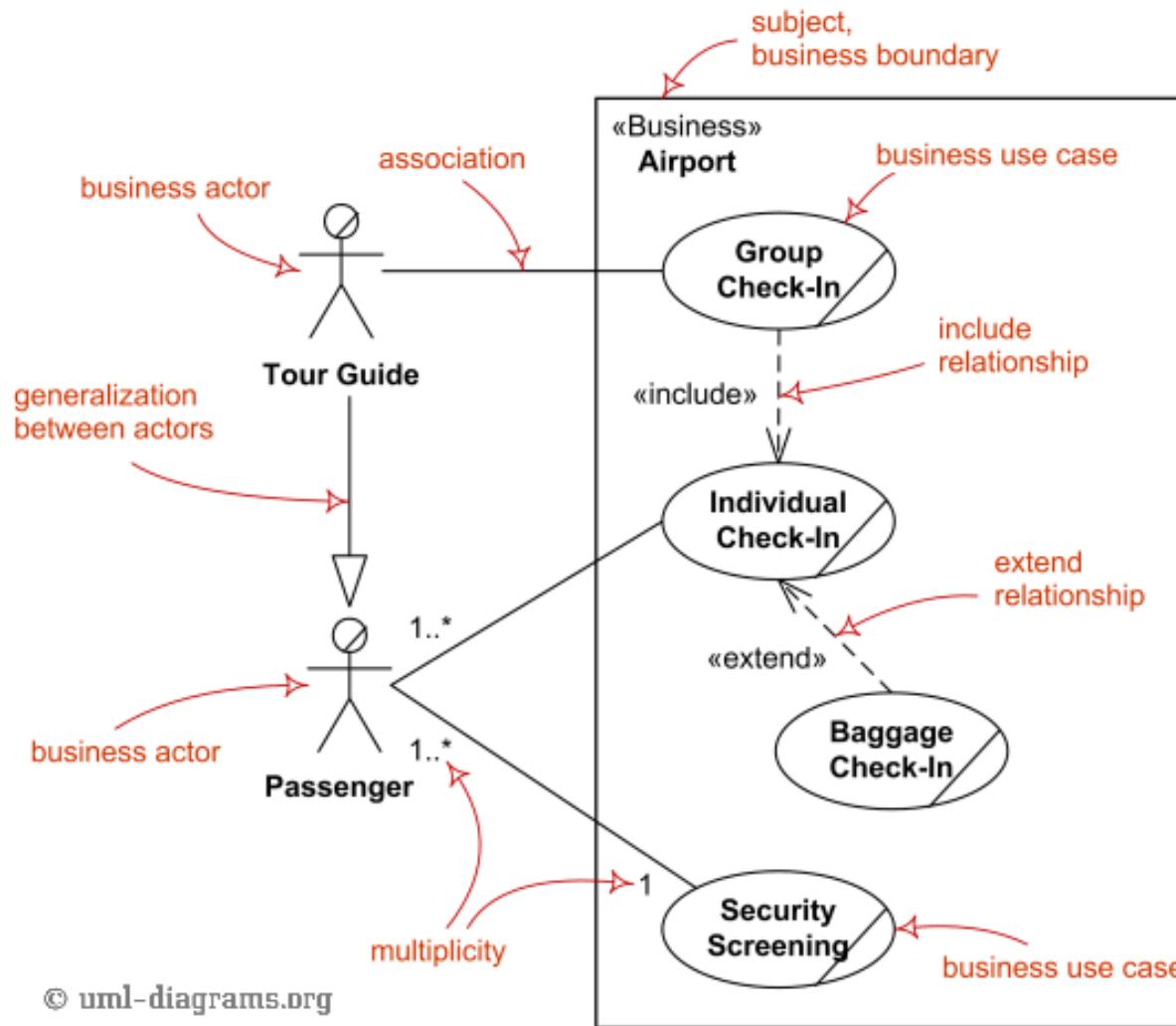
# UML Component Diagram



# UML Sequence Diagram



# UML Use Case Diagram



# More Theory vs. Practice

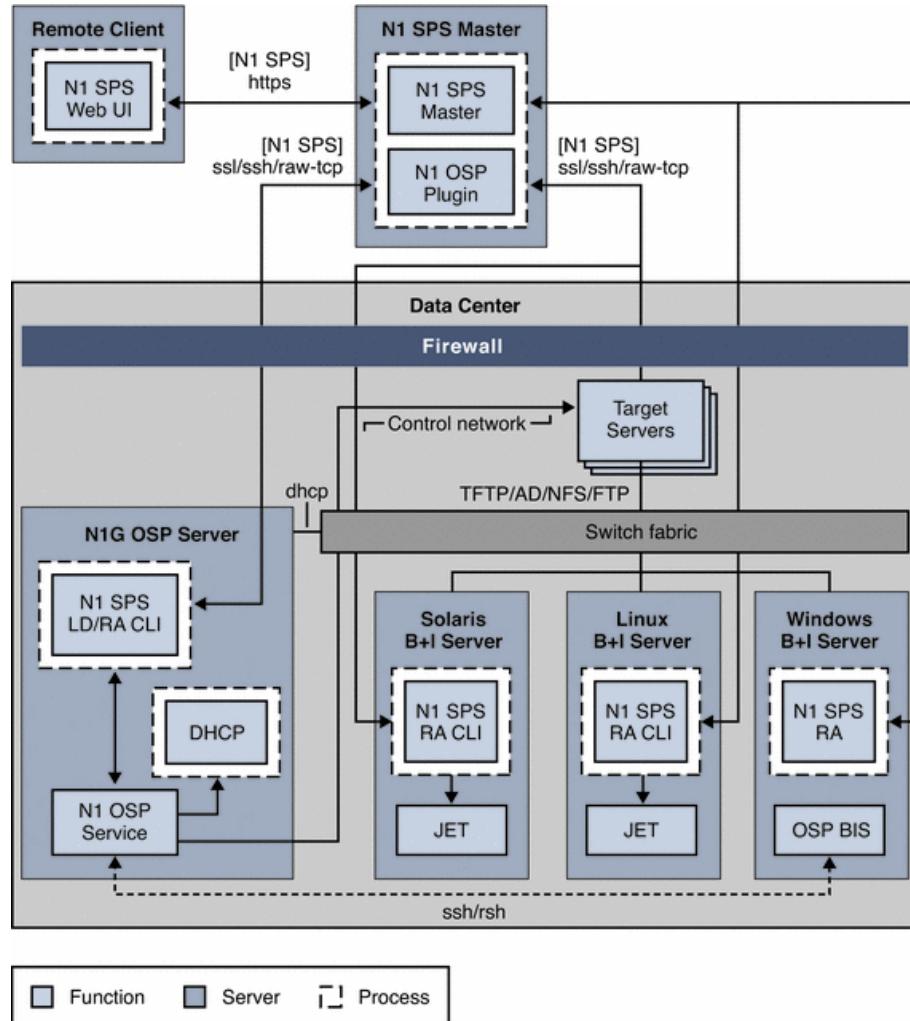
Ok, that is the theory

Let's look at some real examples

Generally:

- Only one diagram; includes hardware and software components
- Boxes: usually don't follow UML
- Lots of lines, arrows
  - Some of which are labeled
- Explanatory text

# Oracle: Sun OS provisioning Plug-in



# Oracle: text accompanying diagram

The figure describes the following relationships among the OS provisioning components and uses slightly abbreviated terminology:

Remote Client – The N1 SPS remote client runs the browser interface and command-line interface. The remote client can be a separate system from the Master Server.

N1 SPS Master – The N1 SPS Master Server is the main processing engine of the N1 SPS software.

N1 OSP Plug-In – The OS provisioning plug-in is installed on the Master Server. The plug-in provides functionality to install operating systems on different hardware platforms that support different protocols.

N1 OSP Server – The OS provisioning control server, usually referred to as the OS provisioning server, is the main processing engine of the OS provisioning plug-in. The OS provisioning server runs the OS provisioning service (N1 OSP Service), which orchestrates the OS provisioning activities. The OS provisioning server controls the target hosts through a control network using appropriate network management protocols (such as IPMI, ALOM, LOM, RSC, ILO, and terminal server). These protocols over the control network are used to automate the power, boot, and console services.

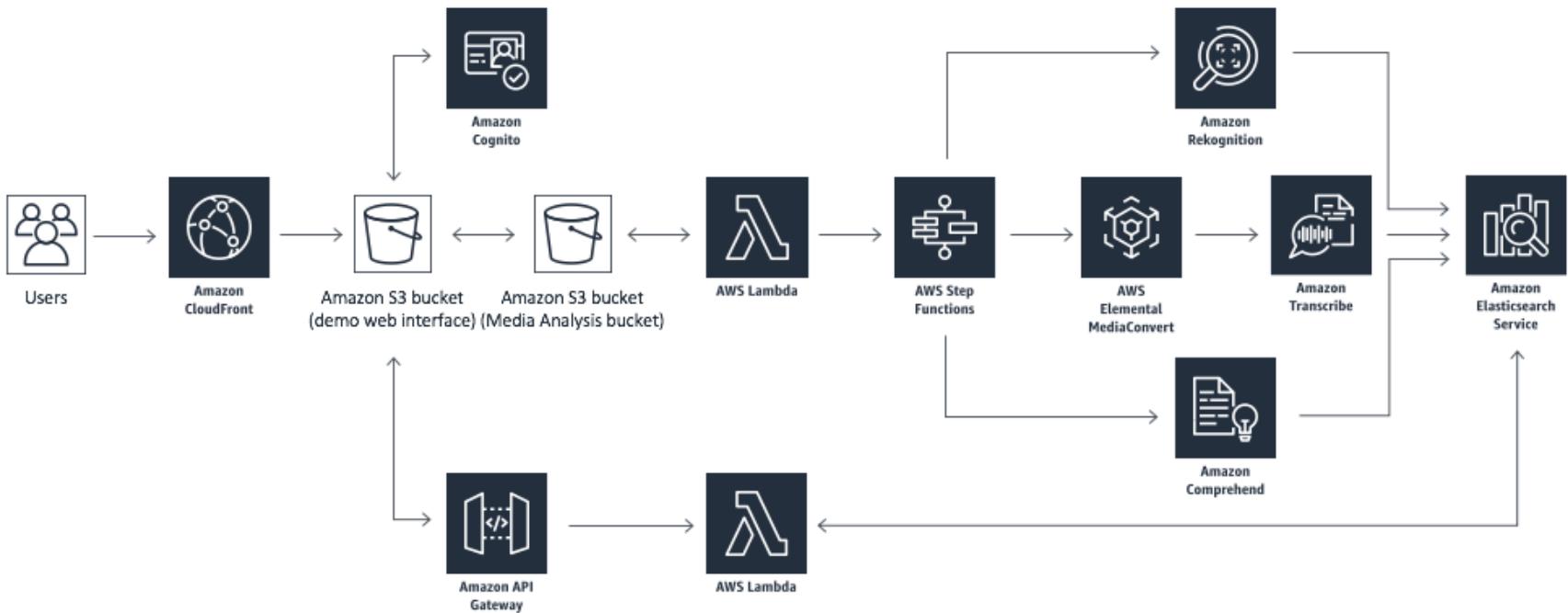
The OS provisioning server supports extensive network topologies (multiple subnets, VLANs, and so on). The OS provisioning server has a bundled DHCP server to serve relevant IP addresses and other boot specific information to target hosts .

Boot and Install Servers – Three servers are shown supporting OS specific Boot and Install servers:

- Solaris B + I Server – The Solaris boot and install server uses the JumpStart™ Enterprise Toolkit (JET) to automate the installation of the Solaris distribution media and installation profile.
- Linux B + I Server – The Linux boot and install server uses the Linux Kickstart technology.
- Windows B + I Server – The Windows boot and install server uses Windows Remote Installation Services (RIS) technology.

The boot and install servers have OS-specific boot and install services for automation and monitoring purposes. You have to set up the Linux and Windows boot and install servers outside of the OS provisioning plug-in. For Linux systems, you have to install the N1 SPS Remote Agent (RA) manually. For Solaris systems, the OS provisioning plug-in installs and configures the RA.

# An architecture using AWS



# Text accompanying previous diagram

(um, there wasn't any)

Big assumption in previous diagram: you know all the AWS components

# In Practice (What you should do)

Use your own style of boxes and lines

- Be clear and consistent
- You may nest boxes inside boxes, but be clear on meaning
- Layout must be intuitive

For connectors:

- Be clear and consistent
- Describe what they mean, maybe have a “legend”
- Connectors are generally not well documented; do better!

Sequence diagrams:

- Follow UML. There are many free tools for it

Text

- Write more than you think you need
- Explain diagrams by using text
- Explain rationale behind important decisions

Use cases

- Follow standard use case style
- Use case maps are often not very useful

# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**

# **Quality Attributes (Non-functional attributes)**

“A non-functional property of a software system is a constraint on the manner in which the system implements and delivers its functionality”

Functional attributes (features): the WHAT

Non-functional attributes (quality attributes): the HOW (how the system behaves)

# Typical Quality Attributes

Security

Reliability

Availability

Efficiency

Performance

Scalability

Fault-tolerance

Usability

Portability

Internal:

- Complexity
- Implementability
- Maintainability

# External or Internal?

Some Quality Attributes have direct impact on the user of the system (External)

Some are invisible to the user; but important to the developers (internal)

Sometimes called “Run time” and “Design time” (the Bass book)

The external ones are generally easier to characterize and quantify.

# What are Quality Attributes

Group interactive activity:

You are wanting to buy a new car.

Name as many features of the car as you can

- Let's group them by:
  - Functional attributes
  - Non-functional attributes

How do you characterize functional attributes versus non-functional (quality) attributes?

# Measuring Quality Attributes

Functional Attributes are often binary measures:

- Does the car have door handles?
- Does the car have 17 cup holders?
- How many seats does it have? (binary: does it have four seats?)

Quality Attributes are usually a scale:

- How fast will your car go?
- How easy is it to use? (Hmm. How do you measure it?)
- How expensive is it to maintain?

# Quality Attributes and Architecture

Quality Attributes are system-wide properties

You often need a uniform approach across all components

You usually need to design for the QA from the beginning of the architecture

- At the very least: think about it

But Quality Attributes also tie closely to the code

# Early in the architecture

(Performance example)

Agile Mantra:

1. Make it
2. Make it right
3. Make it fast

“Premature optimization is the root of all evil.”

Both the above concentrate on the coding aspect, and ignore the necessity of an architecture that supports necessary performance

If your architecture is inherently slow, all the local optimization won't save you

(note: does optimization == performance?)

# Early in the Architecture

(reliability example)

During coding:

- check the return from every function to make sure it performed properly
- Check transactions to make sure they complete
- Make sure you have try-catch blocks at every point that could fail

At architecture:

- What if a function fails? Devise general approaches
- Decide on transaction rollbacks, voting, etc.

# Early in the Architecture

(security example)

## Coding:

- Avoid structures that allow buffer overflow
- Implement intrusion detection
- Implement authentication, authorization, encoding, etc.

## Architecture:

- Which measures do you use?
- Do you have a security layer in the software?
- An Authentication module? Where does it live?

# Early in the Architecture

(security example: Skype)

You are designing Skype. What should you do about security?

Ok, you have password-protected accounts. What if someone forgets their password?

What if the mail to reset their password fails?

Coding: how do you implement that step, given that you are already implementing a trouble ticket system?

# Early in the Architecture

(Availability example: Amazon)

Architecture:

- What is the physical architecture of the replicated system?
  - Hot Spare, Warm Spare, Cold Spare, etc.
  - Full duplication vs. N+1 sparing, etc.
  - Already has multiple servers for load, geographic location, etc.  
Can we use them for availability?
- What are the restart levels, and what do they mean?
  - (How bad does the system have to get before we take a unit out of service?)

Coding:

- Implement “hot spare”
- Implement “escalating restarts”

# Early in the Architecture

(Usability example: Parking app for paid parking)

## Architecture:

- What is the general flow of the user interactions?
- How to identify a car?
  - Is it based on license places? If so, capture or enter them?

## Implementation:

- How should I lay out this screen?
- What font and style?
- What is the flow?

# Early in the Architecture

(portability example: Compiler)

Architecture:

- What are the cost and benefit of being completely portable?
- What does portability mean for this system?
- What modules are custom for a particular machine architecture, and which can be shared? (Basic partitioning of the system!)

Coding:

- Make all the code machine-independent
- AND: make it produce different object code for different machines.

# Early in the Architecture

(scalability example: Amazon, the early days)

Architecture:

- How to allow for scaling
- What does scaling mean, anyway?
  - Which dimensions?
- Upper limits?

Coding:

- Use coding that allows data to grow, connections to grow, etc.

# Early in the Architecture

(maintainability example: Amazon, again)

Architecture:

- Partition the software in ways that are intuitive
- Commonality and Variability analysis
- How can we change one part of the system and integrate it with the rest of the system?
  - And keep the system live during upgrades?

Coding:

- Use good coding practices to keep code clear,

# Quality Attribute Decisions

Decisions about quality attributes:

At the architecture level: tend to be strategic

- Includes decisions about tradeoffs among quality attributes

At the implementation level: tend to be tactical

- “Tactics” are implementations of quality attributes.

Footnote: remember definitions of architecture as a set of decisions? These are important decisions!

# Where it All Starts

Early: as general requirements and architecture are being formulated

- The two influence each other

For each Quality Attribute:

1. What does it mean in this domain?
  - a) Common approach: what ways can this QA fail?
2. How important is it (also relative to others)?
3. What does it do to the architecture?
  - a) E.g., does it require certain hardware or software components?
  - b) E.g., does it encourage (or discourage) certain patterns?

# Determining Importance

How important is a Quality Attribute?

1. Explore the consequences of failure (examples):
  1. Inconvenience
  2. Loss of some time
  3. Loss of some money
  4. Loss of lots of money
  5. Loss of customer trust
  6. Loss of life
2. Consider the likelihood of occurrence
3. Consider difficulty/expense of prevention
4. May consider importance relative to other QAs  
(classical risk analysis)

# Example: TicketBoss

Quality Attribute	What does it mean?	Acceptable level?	How Important?
Security			
Availability			
Reliability			
Performance			
Scalability			
Usability			
Portability			
Maintainability			
Extensibility			
(fill in others)			

# Example: TicketBoss

Quality Attribute	What does it mean?	How Important?
Security	Protect customer data, especially payment info. (no observe) Protect event info. (modification) Roles: customer, admin Protect reservations	Extremely important
Availability	Always can see events, etc. Always can make reservations	6 9's: not important 3 9's: the acceptable threshold
Reliability	Transactions work (rollback) Data integrity Reservations	Very important
Performance	Ticket issuance Response time	
Scalability	How many customers at once How many events (and venues) can we support How many customer profiles? How much history can we have? How many countries do we support?	
Usability	How hard is it to buy a ticket How hard is it to find an event	Very important
Portability	Can we run the application on a different type of server? (maybe)	
Maintainability	How hard is it to maintain the software?	
Extensibility	Events without seats? Restaurant reservations? Airline tickets? Government services? Parking spaces?	
(fill in others)		

# **Part 2**

# Selected QAs

Architectural principles for these QAs

Efficiency

Complexity

Scalability

Adaptability

Dependability

# Efficiency

“Efficiency reflects the system’s ability to meet its performance requirements while minimizing its usage of the resources in its computing environment. A measure of the system’s resource usage economy”

Note: not the same as performance

- Performance is an aspect of efficiency

Performance: the system’s ability to perform required functions within the time needed

- Hard realtime performance
- Soft realtime performance
- Non-realtime performance
- We may not care what resources we consume to do it

However, efficiency can impact performance

# Components and Efficiency

## Keep components small

- Ideally a component follows the Single Responsibility Principle
- But can reduce a component's reusability (if you care)

## Keep Component Interfaces simple and compact

- Follow Interface Segregation Principle
- But not too far: too simple, stripped of context, can mean message processing overhead

## Allow multiple interfaces to the same functionality

- It might eliminate need for adapters

## Separate processing components from data

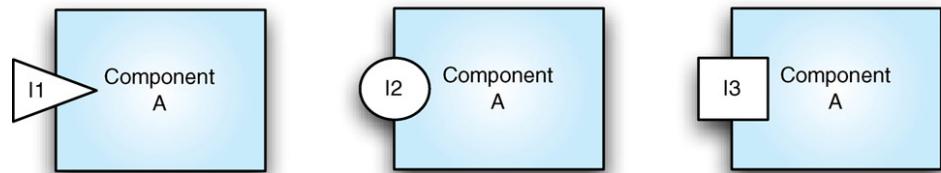
- Can tune either processing or data without affecting the other

## Separate data from metadata

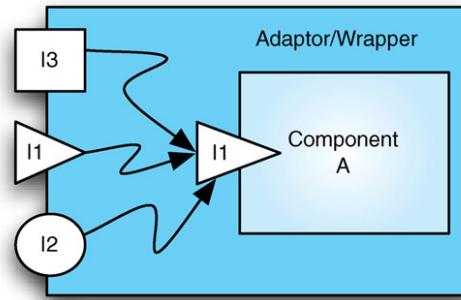
- Data without the metadata is smaller

# Exporting multiple interfaces: approaches

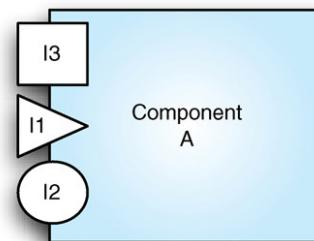
Multiple copies of same component, with different interfaces



A wrapper exposing different interfaces



Component itself has different interfaces



# Efficiency and Software Connectors

Carefully select connectors

- Find those that are the best, simplest fit

Use broadcast connectors with caution

- Broadcast may increase flexibility
- But components may receive data they don't need

Make use of asynchronous interaction whenever possible

- So you don't have components waiting
- But there is overhead with context switching

Use location transparency judiciously

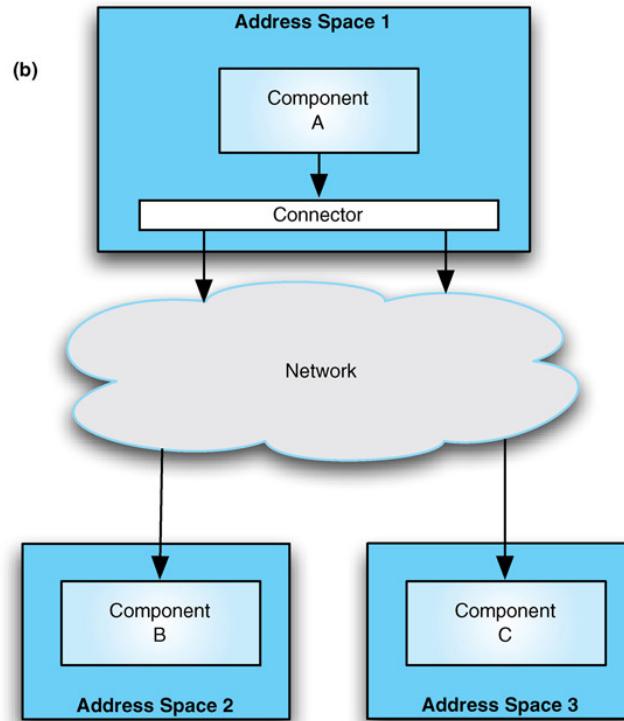
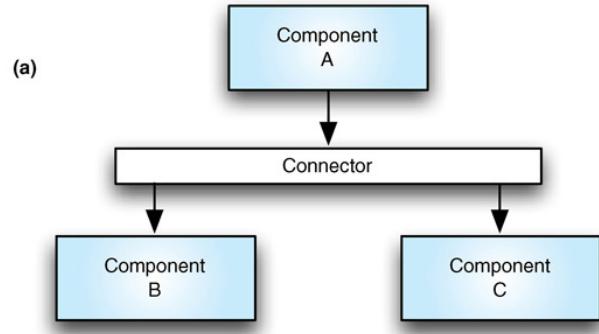
- Distributed components don't have to know their deployment details.
- Flexibility, but may increase overhead
- Among other things, may be difficult to achieve

# Location Transparency

Top: simple connection

Bottom: deployed differently

- Looks the same from outside
- But throughput may suffer



# Efficiency and Architectural Configurations

## Keep Frequently Acting Components Close

- “Close” in what ways? (There are several)
- Reduce number of interactions
- Beware of “pass through” messaging (see picture and System 75 story)

## Carefully select and place connectors in the architecture

- Simple specialized usually more efficient than general purpose
- System 75 (again), Call tree system

# Efficiency: Memory Usage

Some systems (still) have constrained memory

Example: embedded systems

Systems will be small, so architectures will be simple

Numerous sub-architectural patterns (you could call them tactics) in memory-constrained systems.

# Efficiency and Architectural Configurations – Styles and Patterns

- Asynchronous architectures (e.g. Publish-Subscribe) don't work well for realtime systems
- Large repository systems don't work well for memory constrained systems
- Streaming data system MIGHT use event-based architectures, but there is usually a computational penalty
- Layers can introduce gratuitous context switches (inefficient)
- Incremental data delivery: batch doesn't work, pipes and filters usually doesn't work well
- Model-View-Controller not a good fit for distributed/decentralized systems (tight coupling among M, V, & C)
- In Client-Server, the server can become a bottleneck
- Broker can load balance to maximize efficiency (make sure Broker is efficient, though)
- Blackboard: hard to support parallelism
- Presentation Abstraction Control: high overhead among agents
- Microkernel: can have high overhead
- Reflection: Meta-object protocols are often inefficient

## Discuss Efficiency and Architecture in General

Discussion point: For efficiency and most other quality attributes, they don't show up in the architecture as identifiable components

- Possible exceptions: security, availability

# Performance

What is the difference between efficiency and performance?

# Levels of performance

## Levels

- Hard Realtime
- Soft Realtime
- Non-realtime

## Aspects

- Latency
- Response time

Discuss architectural implications for each

Discuss what you might trade off for high performance

- Come up with specific examples

Space

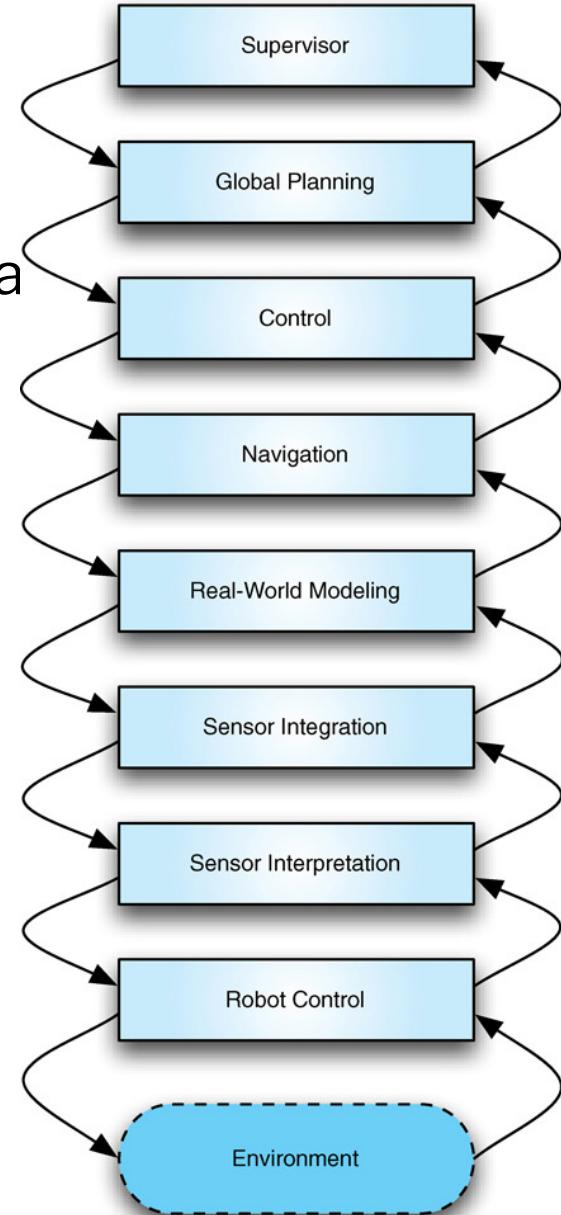
Understandability/Maintainability

Features

Cost (yes, it's a quality attribute)

# Layers

Mobile robot system  
Many layers may slow performance



# Performance: Architectural Considerations

Consider network latency

Consider parallel processing

Might use background processes

Reduce context switches

Generally, increased flexibility means decreased performance

Performance optimization can make code harder to understand, impacting maintainability

# Performance and Patterns

**Layers**: not strong

**Pipes and Filters**: can often optimize for performance.

**Broker**: Not necessarily good, but can use it for load balancing

**Peer to Peer**: Be careful of the broadcast overhead

**Microkernel, Reflection**: Not good

**Event-driven**: might be good, because you can respond to events quickly (usually).

# **Complexity**

# Complexity: definition 1

“Complexity is a system’s property that is proportional to the size of the system, the number of its constituent elements, the size and internal structure of each element, and the number and nature of the elements’ interdependencies.”

- How many parts does it have (relative to its size)?
- How interdependent are the parts?

## Why do we care?

Relative to size: (true, but bigger systems are harder to understand anyway.)

# Complexity, definition 2:

“Complexity is the degree to which a software system has a design or implementation that is difficult to understand and verify.”

- How hard is it to grok?
- Is this always complexity? (Can you have a simple system that is hard to understand?)
- So it's “Understandability”
- But it's really closely tied to complexity (definition1)

Essential complexity

Accidental complexity

(Source: Fred Brooks, “No Silver Bullet”)

# Measuring Complexity

Numerous attempts to measure it

- Most based on the code, or the detailed design
  - McCabe Complexity Metric
  - Halstead metrics (a.k.a. “software science”)
- Architecture complexity measures: usually module-based
  - Coupling
  - Fan-in, fan-out
  - Number, size (of each module), etc.
  - Tools used to analyze code and calculate them

An exact number probably doesn't matter much

We have a pretty good feel for an architecture that is too complex

Discuss architectural implications of complexity  
(either definition)

How do we design to keep complexity low?

# Complexity and Components

Separate concerns into different components

- But don't go overboard, of course

Keep only the functionality inside the components

- Keep the interaction outside the components

Keep the components cohesive

- Single Responsibility Principle

Be aware of the impact of off-the-shelf components on complexity

- They tend to be general purpose, which means greater complexity

Insulate processing components from changes in data formats

- Related to keeping interaction outside components

# Complexity and Connectors

Treat connectors explicitly

- Helps keep interaction out of the components
- Limit different kinds or connectors

Keep only interaction facilities inside connectors

- The functionality goes in the components!

Separate interaction concerns into different connectors

- For example, different connectors for data exchange and for data compression
- Single Responsibility Principle applied to connectors

Restrict interactions facilitated by each connector

- Only involve the components that care about the data you are sending

Be aware of the impact of off-the-shelf connectors on complexity

- Same song, second verse...

# Complexity and Architectural Configurations

## Eliminate unnecessary dependencies

- It's intuitive that greater interdependencies means greater complexity. Two explicit reasons:
  - There are a greater number of interaction paths
  - It's more difficult to control and predict behavior of the system
- See Linux example, next slides

## Manage all dependencies explicitly

- Linux example: note that the documentation of the architecture only mentions some of the dependencies – they are the only ones considered explicitly. (Developers have to discover those dependencies on their own through looking at the code.)

## Use hierarchical decomposition

- Linux example: only seven major chunks: easier to grok

# Complexity and Information Hiding

Discuss

Off-the-shelf components:

- Can hide information!
- Except when they don't!

# Complexity and selected patterns

**Layers**: Easy to understand, but beware of “pass-through” messaging, which may increase complexity

**Pipes and Filters**: Generally pretty simple, if the problem fits it. Note the separation of concerns

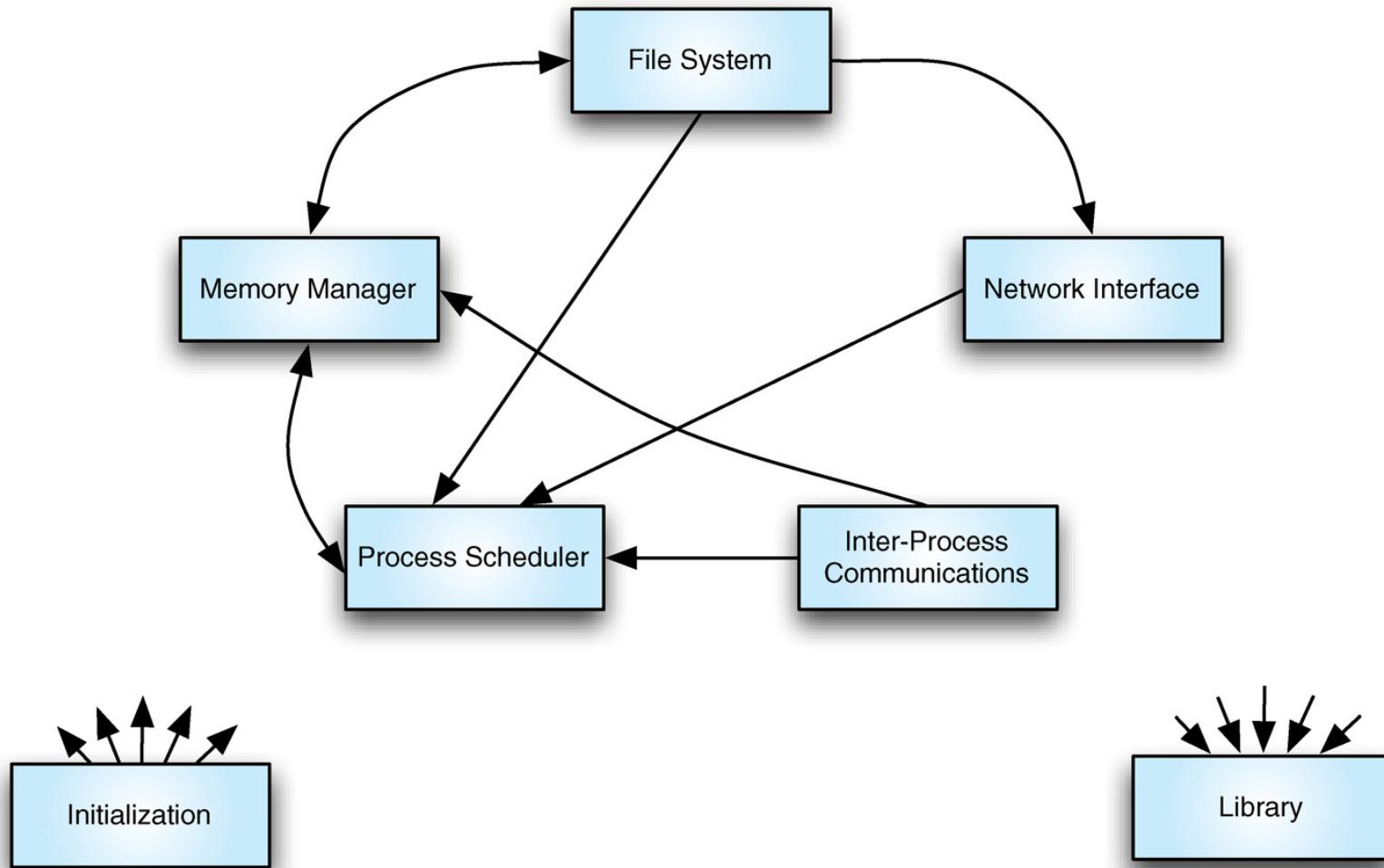
**Model View Controller**: Only three components, but interaction among them is tight and complex.

**Presentation Abstraction Control**: good separation of concerns

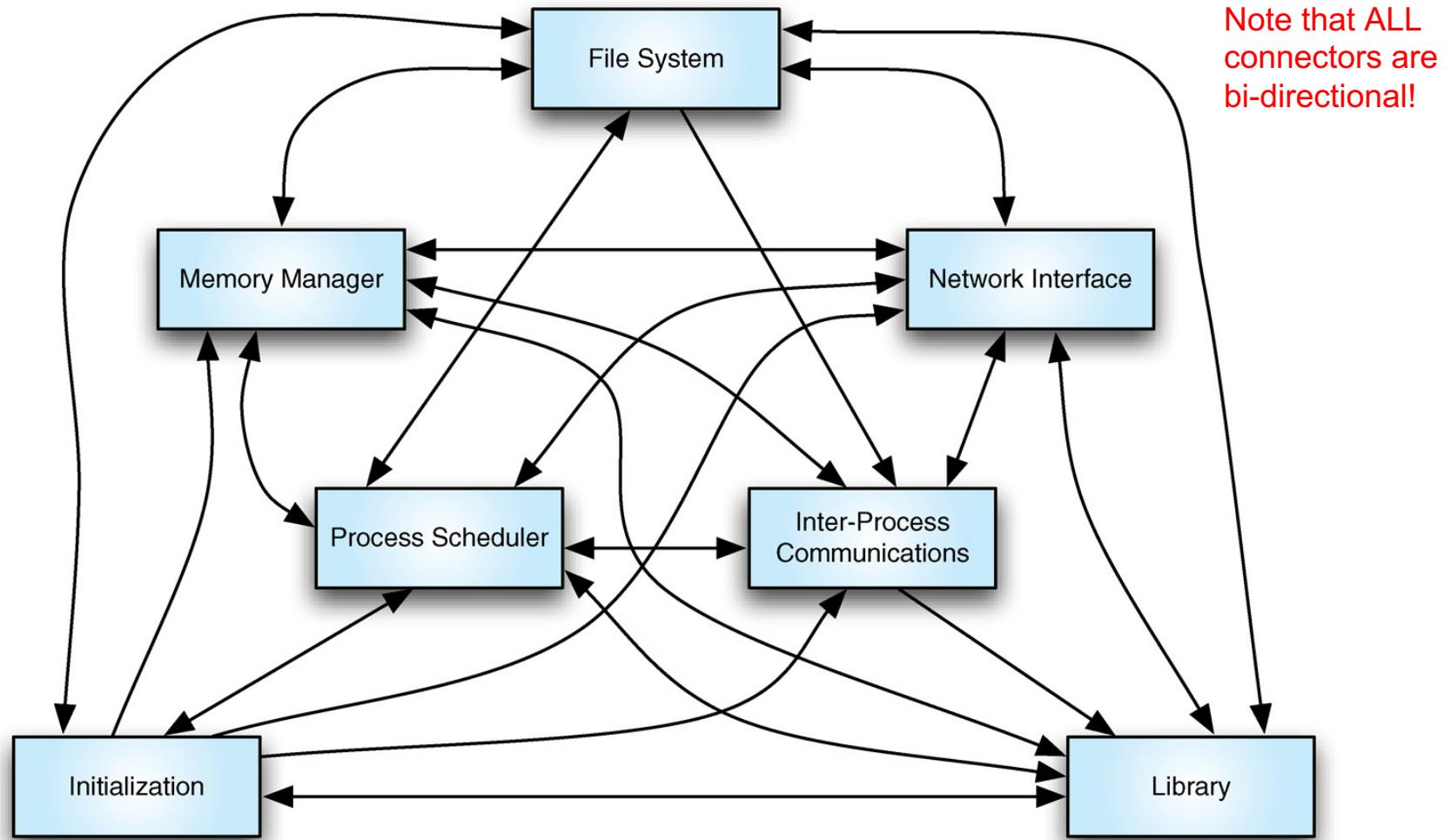
**Microkernel**: can provide a simpler interface to application, hiding implementation details

**Broker**: clean separation of concerns, but more components

# Linux Architecture, as documented



# Linux, as implemented (yikes!)



# How to Increase Complexity (!)

## Speculative Generality

- (But on the other hand: coding unanticipated changes can result in messy code and excess complexity)

Adding Portability (if you don't need it)

Using Patterns (if you don't need them)

Adding Features (that you don't need)

# Scalability and Portability

“Scalability is the capability of a software system to be adapted to meet new requirements of size and scope”

- The interesting cases are usually scaling a system *up*
- Can you give examples?

“Portability is a software system’s ability to execute on multiple platforms (hardware or software) with minimal modifications and without significant degradation in functional or non-functional characteristics”

- The interesting cases are for heterogeneous platforms, of course

# Scalability and Components

Give each component a single, clearly defined purpose

- SRP yet again!
- Allows components to grow or be replicated

Give each component a single, understandable interface

- As above

Do not burden components with interaction responsibilities

- Yet again

Avoid unnecessary heterogeneity

- Keep components compatible with each other

Replicate data when necessary

# Scalability and Connectors

Use explicit connectors (e.g., use specified general interfaces)

- Yet again

Give each connector a clearly defined responsibility

- SRP yet again

Choose the simplest connector suited to the task

Be aware of differences between direct and indirect dependencies

- (Law of Demeter)

Do not place application functionality inside connectors

- Yet again

Leverage explicit connectors to support data scalability

# Scalability and Architectural Configurations

Avoid System bottlenecks

Try to do things in parallel

- Distribute work across multiple machines
- Because much scaling is increasing the number of requests served, it often works well

Place data sources close to data consumers

Try to make distribution transparent

- Homogeneous distribution is generally better for scalability than heterogeneous distribution

# **Homogeneous vs. Heterogeneous Distribution**

Discuss

# Scalability and Patterns

**Layers**: not particularly strong here

**Pipes and Filters**: can sometimes extend easily through parallel processing.

**Broker**: supports scaling very well

**Half-Object Plus Protocol**

- Supports scalability very well
- Not usually considered an architecture pattern, but it has an architectural flavor.

**Peer to Peer**: usually scales well

- But careful of the broadcast overhead

**Publish Subscribe**: good for scalability

**Shared Repository**: should scale easily

# Portability

## The Theory:

- Pick a portable language and portable libraries, and you are in good shape!
- AND: Write your code to be portable, and all is well!

## The Reality:

- It's not that easy!
- Recent news:
  - A chemical research library (written in Python) produced different results when run on different OS (about a 1% difference): The implementation of a python library returned a list of files in a different order.

# Portability and Patterns

Microkernel/Virtual Machine: That's what it's all about

Layers: very strong

Many patterns have separation of concerns that support portability:

- Client-Server, Broker
- MVC, PAC
- Etc.

Other patterns: should generally be neutral

# Maintainability

Supporting factors:

- How easy is it to understand the system (including the code)
- Simplicity (~Complexity)
- Separation of Concerns
- External libraries and other resources used
  
- A lot of sub-architectural design and coding practices

Note that a lot of maintainability is below the level of architecture

# Maintainability and Patterns

**Layers**: strong: separate modification and testing of layers; supports reuse

**Pipes and Filters**: can modify filters (somewhat) independently

**Blackboard**: extendable with independent agents, but difficult to test

**Model View Controller**: tight coupling impedes maintainability

**Presentation Abstraction Control**: separation of concerns supports maintainability

**Microkernel**: easy to extend, separation of concerns

**Broker**: separation of clients from servers allows separate modification

# Dependability

“*Reliability* is the probability that the system will perform its intended functionality under specified design limits, without failure, over a specified time period.”

“*Availability* is the probability that the system is operational at a given time.”

“A software system is *robust* if it is able to respond adequately to unanticipated run time conditions.”

“A software system is *fault-tolerant* if it is able to respond gracefully to failures at run time.”

“*Survivability* is a software system’s ability to resist, recognize, recover from, and adapt to mission-compromising threats.”

“*Safety* denotes the ability of a software system to avoid failures that will result in loss of life, injury, significant damage to property, or destruction of property.”

# **Dependability**

(See Deck Module 06 extra)

# Security

Some security measures can be added after development but is very helpful to design for security right from the beginning.

This discussion provides an overview of security concepts and how they interact with software architecture

# Attacks: One way to look at them

Basic types of attacks on data:

- Access it
- Corrupt (modify) it
- Remove it (special case of modification)
- Prevent access to it (includes removing it)

Basic types of attacks on system resources (e.g., CPU):

- Use it
- Change what someone is running
- Prevent it from running

(Obviously, they all go together)

# **Three main aspects of Security**

Confidentiality

Integrity

Availability

# Confidentiality

Preventing unauthorized parties from accessing information

- Or perhaps even being aware of the existence of the information
- Software systems should protect confidential information from being intercepted.
- System should store sensitive data in a secure way so unauthorized users cannot discover content or existence of such data.
- Data aggregation and partial data:
  - Some data may be public in the aggregate, but private individually (example: census data)
  - Some data may be public in a piece, but private when put together
  - Data protection includes protection from deriving data inferentially

# Integrity

Only authorized parties can manipulate the data

- And only in authorized ways

Analogous constructs at the programming language level:

- Java: private, package, protected, public access levels
- C++: “const” prevents modification

At software architecture level:

- Watch interfaces

Integrity requires that user identity be established

- User authentication process (e.g. username/password)

# Availability

Resources are available if they are accessible by authorized parties on all appropriate occasions

Malicious actions can make resources unavailable (a denial of service (DoS) attack)

Distributed applications may be vulnerable to a distributed denial of service (DDoS) attack.

# Design Principles

## Least Privilege

- Give each component only the privileges it requires

## Fail-safe defaults

- Deny access if explicit permission is absent

## Economy of mechanism

- Adopt simple security mechanisms; avoid doing the same check in different places

## Complete mediation

- All accesses to entities are checked to ensure they are allowed

## Open design

- The security of a mechanism should not depend on the secrecy of its design or implementation

## Separation of Privilege

- Do not grant permission based on a single condition
- Sensitive operations should require the cooperation of more than one key party

## Least common mechanism

- Mechanisms used to access separate resources should not be shared
- The compromise of one resource should not allow the compromise of another

## Psychological Acceptability

- Security mechanisms should match the mental model of the users
- Security shouldn't make it harder to use by legitimate users

## Defense in depth

- Have multiple defensive countermeasures

# Defense in Depth: Microsoft Internet Information Service

POTENTIAL PROBLEM	PROTECTION MECHANISM	DESIGN PRINCIPLES
The underlying dll (ntdll.dll) was not vulnerable because...	Code was made more conservative during the Security Push.	Check precondition
Even if it were vulnerable...	Internet Information Services (IIS) 6.0 is not running by default on Windows Server 2003.	Secure by default
Even if it were running...	IIS 6.0 does not have WebDAV enabled by default.	Secure by default
Even if Web-based Distributed Authoring and Versioning (WebDAV) had been enabled...	The maximum URL length in IIS 6.0 is 16 Kbytes by default (>64 Kbytes needed for the exploit).	Tighten precondition, secure by default
Even if the buffer were large enough...	The process halts rather than executes malicious code due to buffer-overrun detection code inserted by the compiler.	Tighten postcondition, check precondition
Even if there were an exploitable buffer overrun...	It would have occurred in w2wp.exe, which is running as a network service (rather than as administrator).	Least privilege  (Data courtesy of David Aucsmith)

# Access Control Models

## Classic discretionary access control

- A set of subjects that have privileges (permissions)
- A set of objects on which these privileges can be exercised
- An access matrix specifies the privilege a subject has on a particular object
- Commonly implemented as an access control list

## Role-based access control

- Roles become the entities that are authorized with permissions
- Roles can form a hierarchy
- Principals are assigned one or more roles

## Mandatory access control

- Multilevel security environments
- Each subject and each object are assigned a security label

# Connector-Centric Architectural Access Control

## Basic concepts

- Subject: the user on whose behalf a piece of software executes
- Principal: a subject can take upon multiple principals – they encapsulate the credentials that a subject possesses to acquire permissions
- Resource: an entity for which access should be protected.
- Permission: the operations on a resource a component may perform
- Privilege: describes what permissions a component possesses.
- Policy: ties together the above concepts: specifies what privileges a subject, with a given set of principals, should have in order to access resources

# Role of Architectural Connectors

Architectural access control is centered on connectors because they propagate privileges that are necessary for access control decisions

Supply security contract:

- Specifies the privileges and safeguards of an architectural element

Regulate and enforce contract:

- Can determine the subjects for which the connected components are executing
- Also determine whether components have sufficient privileges
- Might provide secure interaction between insecure components

If model is formally specified, you can write an algorithm to check architectural access controls

```
<complexType name="Security.PropertyType">
<sequence>
<element name="subject" type="Subject"/>
<element name="principals" type="Principals"/>
<element name="privileges" type="Privileges"/>
<element name="policies" type="Policies"/>
</sequence>
</complexType>

<complexType name="SecureConnectorType">
<complexContent>
<extension base="ConnectorType">
<sequence>
<element name="security" type="Security.PropertyType"/>
</sequence>
</extension>
</complexContent>
</complexType>

<complexType name="SecureSignature">
<complexContent>
<extension base="Signature">
<sequence>
<element name="safeguards" type="Safeguards"/>
</sequence>
</extension>
</complexContent>
</complexType>
```

# formality

# Access control check algorithm

Input : an outgoing interface, Accessing,  
and an incoming interface, Accessed

Output : grant if the Accessing can access  
the Accessed, deny if the Accessing  
cannot access the Accessed

```
Begin
    if (there is no path between Accessing and Accessed)
        return deny ;
    if (Accessing and Accessed are connected directly)
        DirectAccessing = Accessing;
    else
        DirectAccessing = the element nearest to Accessed in the path;
    Get AccumulatedPrivileges for
        DirectAccessing from the owning element, the type, the containing
        sub-architecture, the complete architecture, and the
        connected elements;
    Get AccumulatedSafeguards for Accessed from the owning element, the
        type, the containing sub-architecture, and the complete architecture;
    Get AccumulatedPolicy for Accessed from similar sources;
    if (AccumulatedPolicy exists)
        if (AccumulatedPolicy grants access)
            return grant ;
        else
            return deny ;
    else
        if (AccumulatedPrivileges contains AccumulatedSafeguards)
            return grant ;
        else
            return deny ;
End ;
```

# Example: Secure Cooperation

Uses the C2 architecture pattern

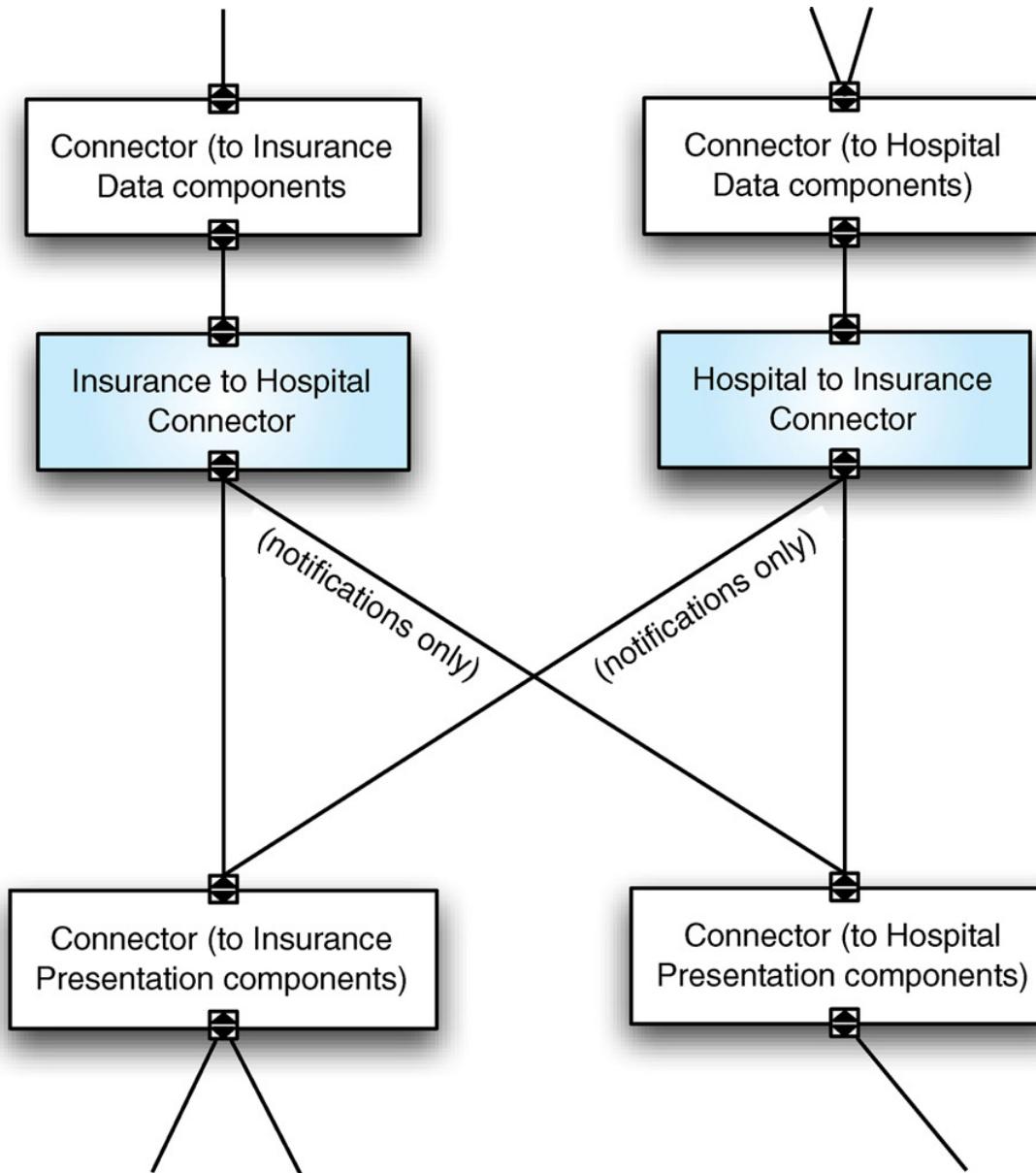
Two parties can share data with each other, but do not necessarily trust each other

- So the data must be subject to control of each party

Two parties: insurance company and hospital

Figure:

- Secure connector on each side



# Trust Management

Decentralized applications, where entities do not have complete information about each other

- Thus, must make local decisions autonomously
- Entities must account for the possibility that other entities may be malicious
- Without a central authority that can manage the entities, each must adopt suitable measures to safeguard itself

It is therefore critical to choose an appropriate trust management scheme for a decentralized application

- And implement appropriate protection measures

Trust: “a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor such an action and in a context in which it affects his own action.”

# Reputation

Closely related to trust

“An expectation about an individual’s behavior, based on information about or observations of its past behavior.”

Trust management systems may use reputation to determine the trustworthiness of an entity (reputation-based systems)

- Several common ones
- May be centralized or decentralized

Example: eBay

- Buyers and sellers rate each other after a transaction
- Centralized reputation management system
- System can be manipulated (any system can): in 2000, peers established positive ratings, then used their reputation to start high-priced auctions, received payment, and disappeared.

# Threats to Decentralized Systems

- Impersonation
- Fraudulent Actions
- Misrepresentation
- Collusion
- Denial of service
- Addition of unknowns
- Deciding whom to trust
- Out-of-band knowledge

# Measures to Address Threats

Authentication

Separation of Internal Beliefs and Externally Reported Information

Making Trust Relationships Explicit

Comparable Trust

# Guidelines to Incorporate into Architecture

Digital Identities

Separation of Internal and External Data

Making Trust Visible

Expression of Trust

Resultant Architectural Style

- Layered architecture works well
- C2 is an event-based layered architecture
- Extention: Practical Architectural style for Composing Egocentric applications (PACE)
  - (one example of an architecture designed to incorporate trust)

# **Software Systems Architecture**

FEUP-M.EIC-ASSO-2023-2024

**Ademar Aguiar, Neil Harrison**

# Dependability

“*Reliability* is the probability that the system will perform its intended functionality under specified design limits, without failure, over a specified time period.”

“*Availability* is the probability that the system is operational at a given time.”

“A software system is *robust* if it is able to respond adequately to unanticipated run time conditions.”

“A software system is *fault-tolerant* if it is able to respond gracefully to failures at run time.”

“*Survivability* is a software system’s ability to resist, recognize, recover from, and adapt to mission-compromising threats.”

“*Safety* denotes the ability of a software system to avoid failures that will result in loss of life, injury, significant damage to property, or destruction of property.”

# Bugs!

What is the difference between a fault and a failure?

A fault is a defect in the system

A failure is incorrect behavior of the system that is visible (to users, for example)

Faults can cause failures

A failure can be caused by different faults

- Of by a combination of faults

A failure may be caused by external events too

Which do we care about?

Both, of course

# Faults (bugs in the code)

Part of our job is to minimize the faults in the code

How?

# Failures

The bottom line

(actually, if a fault NEVER causes a failure, we don't care.)

But we can't guarantee that we eliminated all the faults

- Or eliminated all the faults that will cause failures

And external events can also cause failures

So we have to design our systems to:

- Prevent faults from causing failures
- Prevent external events from causing failures
- Mitigate the effects of failures (that we can't prevent)

# Dependability and Architectural Configurations

Avoid single points of failure

Provide back-ups of critical functionality and data

Support nonintrusive system health monitoring

Support dynamic adaptation

- (maybe)

# Reliability and Patterns

The architecture patterns selected can have a significant impact on the implementation of reliability

Depends on the tactics selected

- Tactics are approaches to implementing aspects of quality attributes

Details covered in chapter 9 discussion, implementation

# Patterns and Reliability

**Peer to Peer**: good for survivability

**Broker**: Can help with availability (But the broker is often a single point of failure)

**Client-Server**: you have to replicate the server (um, and then you might as well use a broker...)

**Layers**: might support some fault tolerance actions

- Not particularly strong for availability (doesn't hurt, doesn't help)

**Pipes and Filters**: not strong

- (but maybe you can have alternate filter paths when one dies)
- For availability of a P&F sequence, you need an independent monitor
- Fault correction can be difficult

**State**: Often central to high availability systems; compatible with it.

# Patterns for Fault Tolerant Software

These are kind of architectural

- But not the structural architecture patterns we have seen
- They can be used with them

Some are overall design approaches

Others are more like tactics

# Units of Mitigation

How can you keep the whole system from being unavailable when an error occurs?

Divide the system into parts that will contain both any errors and the error recovery. Choose the divisions that make sense for your system. Design the rest of the system around these parts that represent the basic units of error mitigation.

- Units perform self-checks
- Units perform their own recovery, where possible
- Units are barriers to errors – prevent errors from propagating to other units

# Units of Mitigation -- examples

## General:

- Repositories
- Broker pattern: each server

## Banking:

- Loans and Accounts should be separate
- Each account might be its own unit (sub-architectural)

## Flight Control:

- Fuel system, each control surface

# Correcting Audits

Faulty data causes errors

Detect and correct data errors as soon as possible.  
Check related date for errors, correct and record the occurrence of the error.

- (“as soon as possible” means before the error causes a failure, of course)

Example:

- Telephone switching systems: endpoint state audits

# **Fail to a Stable State**

# Audits

Some ways data can be checked

Check structural properties

- linked lists are correctly linked
- Pointers into lists, etc., are within bounds

Known correlations

- Cross linkages between different data structures are correct
- Software representing hardware state matches real state

Sanity checks

- Values are in range of expected values
- Checksums correct
- Heap data not corrupted

Direct comparison

- Duplicated data are checked to each other

# Redundancy

How can we reduce the amount of time between error detection and the resumption of normal operation after error recovery?

Provide redundant capabilities that support quick activation to enable error processing to continue in parallel with normal execution.

Note: there are specific tactics for this

# Replication: System Availability

## Duplication strategies

- Full duplication
- $N + M$  sparing (Often:  $N + 1$  sparing)

## Readiness

- Hot Standby
- Warm Standby
- Cold Standby

# Replication: Data Integrity

Data backup

Automated vs. manual

Location: local vs. remote

# Recovery Blocks

How can we make sure that processing results in an error-free value, when executing the same code repeatedly will produce the same error?

Provide a diversity of redundancy implementations, either different designs or different coding. Execute them within a framework that checks for acceptable results from the execution of one and try the next secondary block, if the results were unacceptable.

# Recovery Block, Structure

Ensure: Successful Execution  
By: executing primary block  
Else by: executing secondary block #1  
Else by: executing secondary block #2  
...  
Else by: executing secondary block #n  
Else: trigger exception

The details of each step are design time, not architecture

## Architecture:

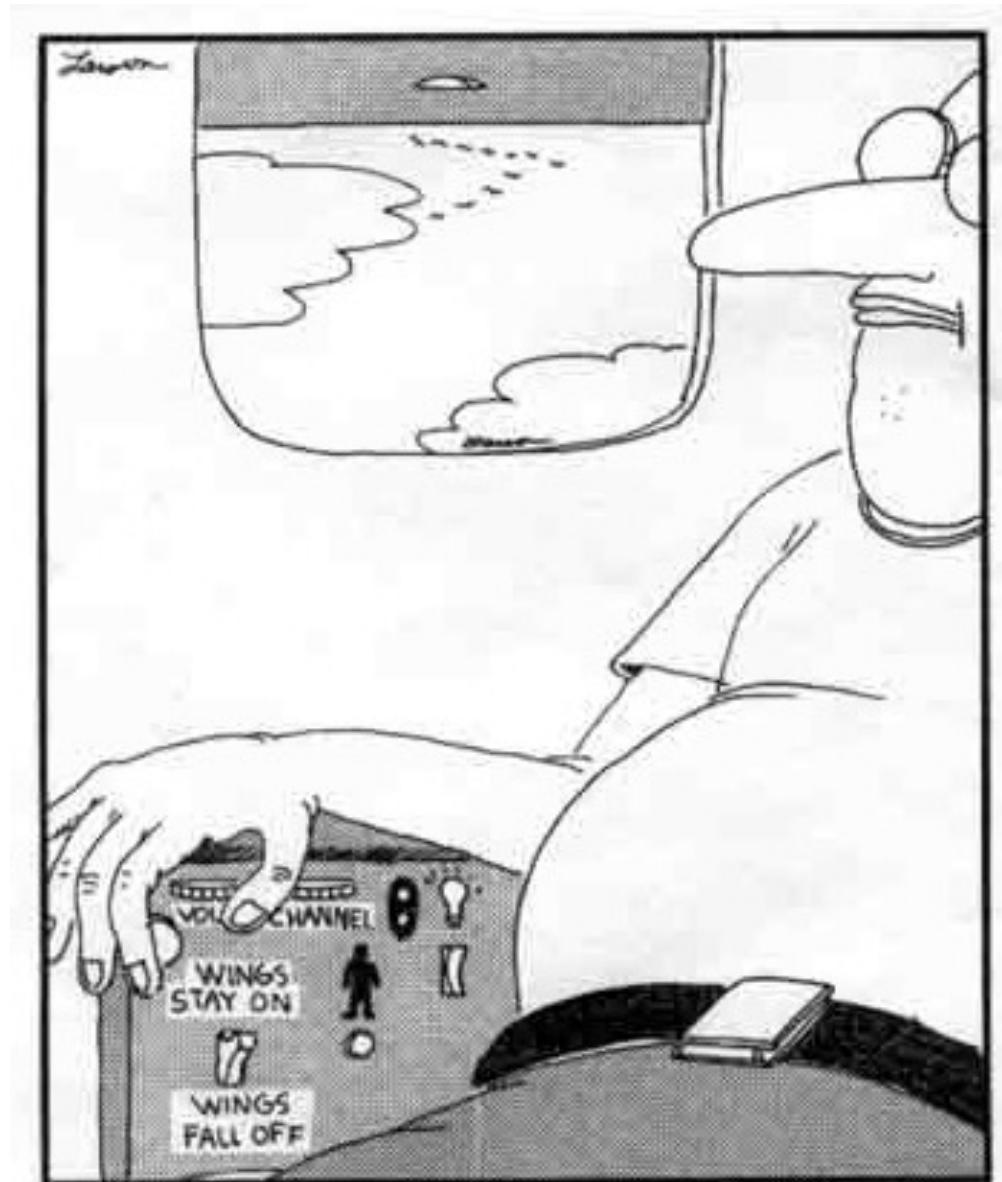
- Decide to use it
- Decide what partitions use it (see also: Units of Mitigation)

# Minimize Human Intervention

How can we prevent people from doing the wrong things and causing errors?

Design the system in a way that it is able to process and resolve errors automatically, before they become failures. This speeds error recovery and reduces the risk of procedural errors contributing to system unavailability.

## Wrong way to do it



Fumbling for his recline button,  
Ted unwittingly instigates a disaster.

# Maximize Human Participation

Should the system ignore people totally? That will reduce procedural errors.

Know the users and their abilities. Design the system to enable knowledgeable operating personnel to participate in a positive way toward error detection and error processing. Provide appropriate Maintenance Interfaces and Fault Observer capabilities to give the operators the information that they need to be able to contribute constructively.

# **Minimize intervention/maximize participation balance:**

Can be tricky

Case study: Boeing 737 Max

Ongoing case study: Autonomous driving systems

# Maintenance Interface

Should maintenance and application requests be intermingled on the application input and output channels?

No (why not?)

Provide a separate interface to the system for the exclusive or almost exclusive use of maintenance interactions.

# Someone in Charge

Anything can go wrong, even during error processing. When this happens the system might stop doing the error processing in addition to not doing the normal processing.

All fault tolerant related activities have some component of the system (“someone”) that is clearly in charge and that has the ability to determine correct completion and the responsibility to take action if it does not complete correctly. If a failure occurs, this component will be sure that the new failure doesn’t stop the system.

# **Someone In Charge**

What are the architectural implications of this principle?

# Someone In Charge

What are the architectural implications of this principle?

- Peer to Peer pattern is not a good fit
- Pipes and Filters:
  - Remember the example we showed?

# Escalation

What does the system do when its attempt to process an error in an component is not achieving the correct effect?

When recovery or mitigation is failing, escalate the action to the next more drastic action.

# Escalating restarts (example)

(Most severe): reboot

Reload: release all memory and reinitialize it.

Cold: release unprotected memory

Warm: preserve memory, but restart all child threads

(Utas, p. 165)

# Fault Observer

The system does not stop when errors are detected, it automatically corrects them. How will people know what faults and errors have been detected and processed, both currently and in the past?

Report all errors to the Fault Observer. The fault observer will ensure that all interested parties receive information about the errors that are occurring.

# Software Update

The system and its applications must not stop operating, not even to install new software.

Design the ability to change your software into its first release. Enhance this capability as necessary in every subsequent release. Do not assume that software will be an easy problem that you can solve after deployment.



# The Big Picture

Dependability aspects require an entire systems approach:

- Hardware
- Software
- Even human behavior

Also may apply to other quality attributes

- For example, human behavior aspects of security

# Case Study: MemPool

System crashed because of memory (heap) corruption

- Most likely used after freed
- Might also have been double deletes

Solution: Object Pools

- Didn't prevent the faults, but mitigated their effect

Tactics used

- Pools per Class (decrease chances of making data illegal)
- Free queue (if used after delete, likely shortly afterward)
- Pointers to free queue separate from the data itself
- Checksum of free objects
- Static pool size

# Handling the Unusual Cases

Think Use Case Extensions

On Steroids. ☺

For inputs to the system:

- Consider the extreme cases
- Consider the cases that appear to be impossible (don't automatically dismiss them)

# Examples of Extreme cases

Oakland earthquake, 1989

Testing our phones

Wall street phone outage

First landing on the moon

Buffer overflows (security issue)

#5ESS: what if a single bit in hardware goes bad?

Y2K Scare

Zug Island...

Beware of:

- “The users would NEVER do that!”
- That event (or combination of events) could NEVER happen!”



# Risk

Do we have to handle EVERY thing that might go wrong?

We have to balance:

- Probability of occurrence
- Frequency of occurrence
- Consequences when it happens
- Cost & time to implement countermeasures
- Ability to implement countermeasures

If time, we will discuss quality attribute analysis methods

# Group Discussion

## Patriot Missile Defense System

What things can go wrong?

- Which architectural approaches might be useful? (See summary list)

Notes:

- We won't go into specific tactics
- Notable failure in 1991; consider it, but think about the whole system needs



# **Summary architectural approaches for reliability**

Units of Mitigation

Correcting Audits

Redundancy

Recovery Blocks

Minimize Human Intervention

Maximize Human Participation

Maintenance Interface

Someone in Charge

Escalation

Fault Observer

Software Update